NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE IMPLEMENTATION OF A
ENTITY-RELATIONSHIP INTERFACE
FOR A MULTI-LINGUAL DATABASE SYSTEM

by

Jacob A. Anthony III
and
Alfred J. Billings

December 1985

Thesis Advisor: David K. Hsiao

Approved for public release; distribution is unlimited

TY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

PORT SECURITY CLASSIFICATION

1b. RESTRICTIVE MARKINGS

CURITY CLASSIFICATION AUTHORITY

CLASSIFICATION / DOWNGRADING SCHEDULE

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release; distribution is unlimited

FORMING ORGANIZATION REPORT NUMBER(S)

5. MONITORING ORGANIZATION REPORT NUMBER(S)

AME OF PERFORMING ORGANIZATION
al Postgraduate School

6b. OFFICE SYMBOL (If applicable)
52

7a. NAME OF MONITORING ORGANIZATION
Naval Postgraduate School

DRESS (City, State, and ZIP Code)
terey, CA 93943-5100

7b. ADDRESS (City, State, and ZIP Code)
Monterey, CA 93943-5100

ME OF FUNDING / SPONSORING GANIZATION

8b. OFFICE SYMBOL (If applicable)

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

DRESS (City, State, and ZIP Code)

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

LE (Include Security Classification)
IMPLEMENTATION OF A ENTITY-RELATIONSHIP INTERFACE FOR THE MULTI-LINGUAL ABASE SYSTEM

RSONAL AUTHOR(S)
ob A. Anthony III & Alfred J. Billings

YPE OF REPORT
ter's Thesis

13b. TIME COVERED
FROM ________ TO ________

14. DATE OF REPORT (Year, Month, Day)
1985 December

15 PAGE COUNT
156

PPLEMENTARY NOTATION

COSATI CODES		
ELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
MLDS - Multi-Lingual Database System, MBDS - Multi-Backend Database System, Entity-Relationship Data Model, Daplex, ABDL - Attribute-Based (Cont)

STRACT (Continue on reverse if necessary and identify by block number)

raditionally, the design and implementation of a conventional database ystem begins with the choice of a data model followed by the specifica- ion of a model-based data language. Thus, the database system is res- ricted to a single data model and a specific data language. An alter- ative to this traditional approach to database-system development is the ulti-lingual database system (MLDS). This alternative approach enables he user to access and manage a large collection of databases via several ata models and their corresponding data languages without the afore- entioned restriction.

n this thesis we present the implementation of a entity-relationship/ aplex language interface for the MLDS. Specifically, we present the mplementation of an interface which translates Daplex language calls into ttribute-based data language (ABDL) requests. We describe the (cont)

STRIBUTION / AVAILABILITY OF ABSTRACT
NCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

AME OF RESPONSIBLE INDIVIDUAL
of. D. K. Hsiao

22b. TELEPHONE (Include Area Code)
408-646-2253

22c. OFFICE SYMBOL
52Hq

DRM 1473, 84 MAR

83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

18. SUBJECT TERMS (Continued)

Data Language, Language Interface

19. ABSTRACT (Continued)

software engineering aspects of our implementation and an overview of the five modules which comprise our entity-relationship/Daplex language interface.

The Implementation of a Entity-Relationship Interface for the Multi-Lingual Database System

by

Jacob A. Anthony, III
Lieutenant, United States Navy
B.S., Pennsylvania State University, 1977

and

Alfred J. Billings
Lieutenant, United States Navy
B.S., University of Utah, 1977

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

December 1985

ABSTRACT

Traditionally, the design and implementation of a conventional database system begins with the choice of a data model followed by the specification of a model-based data language. Thus, the database system is restricted to a single data model and a specific data language. An alternative to this traditional approach to database-system development is the multi-lingual database system (MLDS). This alternative approach enables the user to access and manage a large collection of databases via several data models and their corresponding data languages without the aforementioned restriction.

In this thesis we present the implementation of a entity-relationship/Daplex language interface for the MLDS. Specifically, we present the implementation of an interface which translates Daplex language calls into attribute-based data language (ABDL) requests. We describe the software engineering aspects of our implementation and an overview of the five modules which comprise our entity-relationship/Daplex language interface.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. MOTIVATION

During the past twenty years database systems have been designed and implemented using what we refer to as the traditional approach. The first step in the traditional approach involves choosing a data model. Candidate data models include the hierarchical data model, the relational data model, the network data model, the entity-relationship data model, or the attribute-based data model to name a few. The second step specifies a model-based data language, e.g., DL/I for the hierarchical data model, or Daplex for the entity-relationship data model.

A number of database systems have been developed using this methodology. For example, IBM has introduced the Information Management System (IMS) in the sixties, which supports the hierarchical data model and the hierarchical-model-based data language, Data Language I (DL/I). Sperry Univac has introduced the DMS-1100 in the early seventies, which supports the network data model and the network-model-based data language, CODASYL Data Manipulation Language (CODASYL-DML). More recently, there has been IBM's introduction of the SQL/Data System which supports the relational model and the relational-model-based data language, Structured English Query Language (SQL). The result of this traditional approach to database system development is a homogeneous database system that restricts the user to a single data model and a specific model-based data language.

An unconventional approach to database system development, referred to as the *Multi-lingual Database System* (MLDS) [Ref. 1], alleviates the aforementioned restriction. This new system affords the user the ability to access and manage a large collection of databases via several data models and their corresponding data languages. The design goals of the MLDS involve developing a system that is accessible via four different interfaces, the hierarchical/DL/I, relational/SQL, network/DML, and entity-relationship/Daplex interfaces.

There are several advantages in developing such a system. Perhaps the most practical of these involves the reusability of database transactions developed on an existing database system. In MLDS, there is no need for the user to convert a transaction from one data language to another. MLDS permits the running of database transactions written in different data languages. Therefore, the user does not have to perform either manual or automated translation of existing transactions in order to execute a transaction in MLDS. MLDS provides the same results even if the data language of the transaction originates at a different database system.

A second advantage deals with the economy and effectiveness of hardware upgrade. Frequently, the hardware supporting the database system is upgraded because of technological advancements or system demand. With the traditional approach, this type of hardware upgrade has to be provided for all of the different database systems in use, so that all of the users may experience system performance improvements. This is not the case in MLDS, where only the upgrade of a single system is necessary. In MLDS, the benefits of a hardware upgrade are uniformly distributed across all users, despite their use of different models and data languages.

Thirdly, a multi-lingual database system allows users to explore the desirable features of the different data models and then use these features to better support their applications. This is possible because MLDS supports a variety of databases structured in any of the well-known data models.

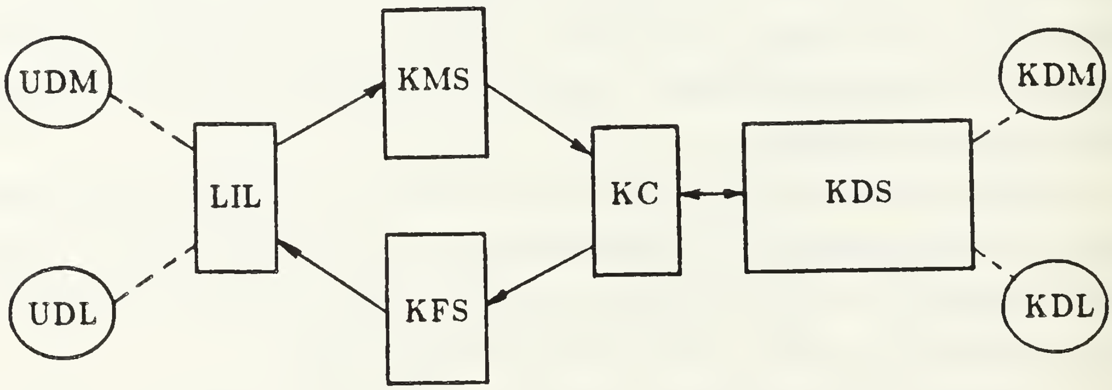
It is apparent that there exists ample motivation to develop a multi-lingual database system with many data model/data language interfaces. In this thesis, an entity-relationship/Daplex MLDS interface is developed.

B. THE MULTI-LINGUAL DATABASE SYSTEM

A detailed discussion of each of the components of MLDS is provided in subsequent chapters. In this section we provide an overview of the organization of MLDS. This assists the reader in understanding how the different components of MLDS are related.

Figure 1 shows the system structure of a multi-lingual database system. The user interacts with the system through the *language interface layer (LIL)*, using a

chosen *user data model (UDM)* to issue transactions written in a corresponding model-based *user data language (UDL)*. LIL routes the user transactions to the *kernel mapping system (KMS)*. KMS performs one of two possible tasks. First, KMS transforms a UDM-based database definition to a database definition of the *kernel data model (KDM)*, when the user specifies that a new database is to be created. When the user specifies that UDL transaction is to be executed, KMS translates UDL transaction to a transaction in the *kernel data language (KDL)*. In the first task, KMS forwards KDM data definition to the *kernel controller (KC)*. KC, in turn, sends KDM database definition to the *kernel database system (KDS)*. When KDS is finished with processing KDM database definition, it informs KC. KC then notifies the user, via LIL, that the database definition has been processed and that loading of the database records may begin. In the



UDM	: User Data Model
UDL	: User Data Language
LIL	: Language Interface Layer
KMS	: Kernel Mapping System
KC	: Kernel Controller
KFS	: Kernel Formatting System
KDM	: Kernel Data Model
KDL	: Kernel Data Language
KDS	: Kernel Database System

Figure 1. The Multi-Lingual Database System.

second task, KMS sends KDL transactions to KC. When KC receives KDL transactions, it forwards them to KDS for execution. Upon completion, KDS sends the results in the KDM form back to KC. KC routes the results to the *kernel formatting system (KFS)*. KFS reformats the results from the KDM form to the UDM form. KFS then displays the results in the correct UDM form via LIL.

The four modules, LIL, KMS, KC, and KFS, are collectively known as the *language interface*. Four similar modules are required for each other language interface of the MLDS. For example, there are four sets of these modules where one set is for the hierarchical/DL/I language interface, one for the relational/SQL language interface, one for the network/DML language interface, and one for the entity-relationship/Daplex language interface. However, if the user writes the transaction in the native mode (i.e., in KDL), there is no need for an interface.

In our implementation of the entity-relationship/Daplex language interface, we develop the code for the four modules. However, we do not integrate these modules with the KDS as shown in Figure 1. The Laboratory of Database Systems Research at the Naval Postgraduate School has procurred the new computer equipment for the KDS. When the equipment is installed, the KDS is to be ported over to the new equipment. The MLDS software is then to be integrated with the KDS. Although not a very difficult undertaking, it is neverless outside the focus of this thesis.

C. THE KERNEL DATA MODEL AND LANGUAGE

The choice of a kernel data model and a kernel data language is the key decision in the development of a multi-lingual database system. The overriding question, when making such a choice, is whether the kernel data model and kernel data language is capable of supporting the required data-model transformations and data-language translations for the language interfaces.

The attribute-based data model proposed by Hsiao [Ref. 2], extended by Wong [Ref. 3], and studied by Rothnie [Ref. 4], along with the attribute-based data language (ABDL), defined by Banerjee [Ref. 5], have been shown to be acceptable candidates for the kernel data model and kernel data language, respectively.

Why is the determination of a kernel data model and kernel data language so important for MLDS? No matter how multi-lingual MLDS may be, if the underlying database system (i.e., KDS) is slow and inefficient, then the interfaces may be rendered useless and untimely. Hence, it is important that the kernel data model and kernel language be supported by a high-performance and great-capacity database system. Currently, only the attribute-based data model and the attribute-based data language are supported by such a system. This system is the multi-backend database system (MBDS) [Ref. 1].

D. THE MULTI-BACKEND DATABASE SYSTEM

The multi-backend database system (MBDS) has been designed to overcome the performance problems and upgrade issues related to the traditional approach of database system design. This goal is realized through the utilization of multiple-backends connected in a parallel fashion. These backends have identical hardware, replicated software, and their own disk systems. In a multiple-backend configuration, there is a backend controller, which is responsible for supervising the execution of database transactions and for interfacing with the hosts and users. The backends perform the database operations with the database stored on the disk system of the backends. The controller and backends are connected by a communication bus. Users access the system through either the hosts or the controller directly (see Figure 2).

Performance gains are realized by increasing the number of backends. If the size of the database and the size of the responses to the transactions remain constant, then MBDS produces a reciprocal decrease in the response times for the user transactions when the number of backends is increased. On the other hand, if the number of backends is increased proportionally with the increase in databases and responses, then MBDS produces invariant response times for the same transactions. A more detailed discussion of MBDS is found in [Ref. 6].

E. THESIS OVERVIEW

The organization of our thesis is as follows: In Chapter II, we discuss the software engineering aspects of our implementation. This includes a discussion of our design approach, as well as a review of the global data structures used for the

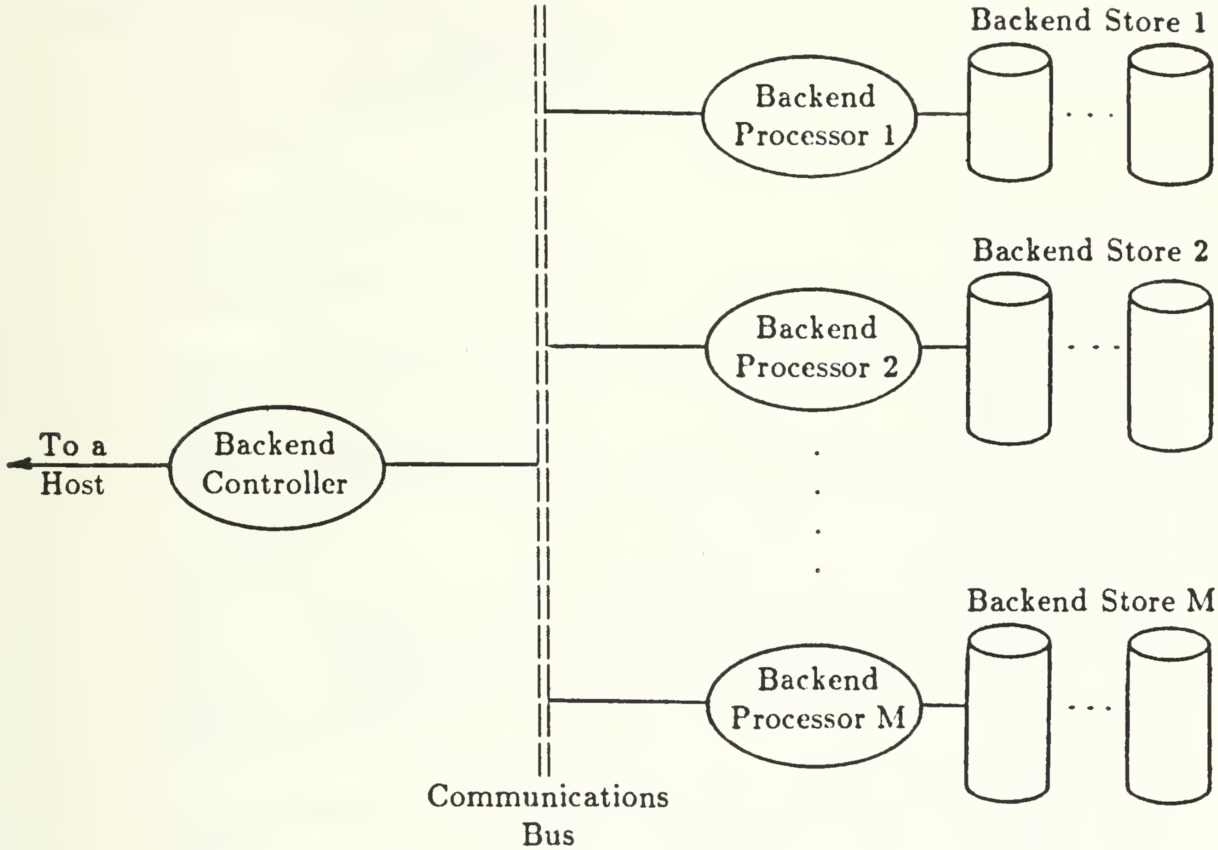


Figure 2. The Multi-Backend Database System.

implementation. Chapter III discusses the storage and creation of the Daplex schemas. In Chapter IV, we outline the functionality of the language interface layer. In Chapter V, we articulate the processes constituting the kernel mapping system. In Chapter VI, we conclude the thesis.

Appendix A contains the data structure used for the interface, and Appendix B provides the modules for the storage and retrieval of the Daplex schemas. The detailed specifications of the interface modules (i.e., LIL and KMS) are given in Appendices C and D respectively. The specifications of the source data language, Daplex, and the target data language, ABDL, are found in [Ref. 7] and [Ref. 8], respectively. Throughout this thesis, we provide examples of Daplex requests and their translated ABDL equivalents. All examples involving database

operations presented in this thesis are based on the university database described in the Daplex User's Manual [Ref. 7], and shown in Figure 3.

```
DATABASE university IS

  TYPE person;
  SUBTYPE employee;
  SUBTYPE support_staff;
  SUBTYPE faculty;
  SUBTYPE student;
  SUBTYPE graduate;
  SUBTYPE undergraduate;
  TYPE course;
  TYPE department;
  TYPE enrollment;
  TYPE rank_name IS (assistant,associate,full);
  TYPE semester_name IS (fall, spring, summer);
  TYPE grade_point IS FLOAT RANGE 0.0 .. 4.0;

  TYPE person IS
    ENTITY
      name : STRING (1 .. 25);
      ssn  : STRING (1 .. 9) := "000000000";
    END ENTITY;

  SUBTYPE employee IS person
    ENTITY
      home_address  : STRING (1 .. 50);
      office        : STRING (1 .. 8);
      phones        : SET OF STRING (1 .. 7);
      salary        : FLOAT;
      dependents    : INTEGER RANGE 0 .. 10 ;
    END ENTITY;

  SUBTYPE support_staff IS employee
    ENTITY
      supervisor    : employee WITHNULL;
      full_time     : BOOLEAN;
    END ENTITY;

  SUBTYPE faculty IS employee
    ENTITY
      rank          : rank_name;
      teaching      : SET OF course;
      tenure        : BOOLEAN := FALSE;
      dept          : department;
    END ENTITY;
```

```
SUBTYPE student IS person
  ENTITY
    advisor       : faculty WITHNULL;
    major         : department;
    enrollments   : SET OF enrollment;
  END ENTITY;

SUBTYPE graduate IS student
  ENTITY
    advisory_committee  : SET OF faculty;
  END ENTITY;

SUBTYPE undergraduate IS student
  ENTITY
    gpa          : grade_point := 0.0;
    year         : INTEGER_RANGE 1 .. 4 := 1;
  END ENTITY;

TYPE course IS
  ENTITY
    title        : STRING (1 .. 10);
    dept         : department;
    semester     : semester_name;
    credits      : INTEGER;
  END ENTITY;

TYPE department IS
  ENTITY
    name         : STRING (1 .. 20);
    head         : faculty WITHNULL;
  END ENTITY;

TYPE enrollment IS
  ENTITY
    class        : course;
    grade        : grade_point:
  END ENTITY;

UNIQUE ssn WITHIN person;
UNIQUE name WITHIN department;
UNIQUE title, semester WITHIN course;
OVERLAP graduate WITH faculty;

END university;
```

Figure 3. The University Database.

II. SOFTWARE ENGINEERING OF A LANGUAGE INTERFACE

In this chapter, we discuss the various software engineering aspects of developing a language interface. First, we describe our design goals, and then outline the design approach that we have taken to implement the interface. Included in this section are discussions of our implementation strategy, our software development techniques, and the salient characteristics of the language interface software. Next, a critique of our implementation is provided, and then we describe the data structures used in the interface. Finally, we provide an organizational description of the next four chapters.

A. DESIGN GOALS

We are motivated to implement a Daplex interface for MLDS using MBDS as the kernel database system, the attribute-based data model as the kernel data model, and the attribute-based data language, ABDL, as the kernal data language. It is important to note that we do not propose changes to the kernel database system or language. Instead, our implementation resides entirely in the host computer. All user transactions in Daplex are processed in the Daplex interface. MBDS continues to receive and process requests in the syntax and semantics of ABDL.

In addition, our interface will be transparent to the user. For example, an employee in a corporate environment with previous Daplex experience could log into our system, issue a Daplex request and receive resultant data in an entity-relationship format. The employee requires no training in ABDL or MBDS procedures prior to utilizing the system.

B. AN APPROACH TO THE DESIGN

1. The Implementation Strategy

There are a number of different strategies we might have employed in the implementation of the Daplex language interface. For example, there is the build-it-twice full-prototype approach, the level-by-level top-down approach, the

incremental development approach, and the advancemanship approach [Ref. 9: pp. 41-46]. We have predicated our choice on minimizing the "software-crisis" as described by Boehm [Ref. 9: pp. 14-31].

The strategy we have decided upon is the level-by-level top-down approach. Our choice is based on first, a time constraint. The interface has to be developed in approximately two quarters. Second, the level-by-level top-down approach lends itself to the natural evolution of the interface. The system is initially thought of as a "black box" (see Figure 1 again) that accepts Daplex transactions and then returns the appropriate results. The "black box" is then decomposed into its four modules, LIL, KMS, KC, and KFS. These modules, in turn, are further decomposed into the necessary functions and procedures to accomplish the appropriate tasks.

2. Techniques for Software Development

In order to achieve our design goals, it is important to employ effective software engineering techniques during all phases of the software development life-cycle. These phases, as defined by Ledthrum [Ref. 10: p. 27], are as follows:

(1) Requirements Specification - This phase involves stating the purpose of the software: "what" is to be done, not "how" it is to be done.

(2) Design - During this phase an algorithm is devised to carry out the specification produced in the previous phase. That is, "how" to implement the system is specified during this phase.

(3) Coding - In this phase, the design is translated into a programming language.

(4) Validation - During this phase, it is ensured that the developed system functions as originally intended. That is, it is verified that the system actually does what it is supposed to do.

The first phase of the life-cycle has already been performed. The research done by Demurjian and Hsiao [Ref. 1] has described the motivation, goals, and structure of MLDS. The research conducted by Goisman [Ref. 11] has extended this work to describe in detail the purpose and design of the Daplex interface. Accordingly, the requirements specifications are derived from the above research.

The system implementation methodology was essentially accomplished and proven during the implementation of DL/I and SQL into MLDS [Refs. 12 and 13]. Our task was to adapt the DL/I and SQL approaches as necessary for the Daplex implementation.

We have used the C programming language [Ref. 14] to translate the design into executable code. Initially, we were not conversant in the language. However, the simple syntax of C and our background in structured languages has made C relatively easy for us to learn.

The main advantage of C is the programming environment in which it resides, the UNIX operating system. This environment has permitted us to partition the Daplex interface, and then manage these parts in an effective and efficient manner. The primary disadvantage to the use of C is that the poor error diagnostics presented by the C compiler can and at times did make debugging difficult. There is an on-line debugger available in UNIX for use with C, but we chose to use conditional computation and diagnostic print statements to aid in the debugging process. To validate our system we have used path testing [Ref 15], a traditional testing technique. We have checked boundary cases, and we have tested those cases considered "normal". It is noteworthy to mention that testing does not prove the system correct, but may only indicate the absence of problems with the cases that have been tested.

3. Characteristics of the Interface Software

We realize that in order for the Daplex interface to be successful, that it must be well designed and well structured. Further, we are cognizant of certain characteristics that the interface must possess. Specifically, it must be simple, and easily read and understood.

The ease with which the code can be understood is vital to keeping the program maintenance effort low. As reported by Fairley [Ref. 16: p. 82], roughly 60% of all software life-cycle costs are incurred after the software becomes operational, so it is important that a maintenance programmer can easily grasp the functionality of the Daplex interface and the relationship between it and the other portions of the system.

We have made every effort to ensure that the C code we have written has these characteristics. For instance, we have avoided the use of the shorthand notations available in C and have used the more readable, and therefor longer version of C whenever possible. This extra code has often made the difference between comprehensible code and cryptic notations. Further, the interface software does not have any hidden side-effects that could pose problems months or years from now. As a matter of fact, we have intentionally minimized the interaction between procedures to ease the burden of maintainability.

In addition to the above software engineering techniques, we require programmers to update documentation of the interface code when changes are made. Hence, maintenance programmers have current documentation at all times, and the problem of trying to identify the functionality of a program with dated documentation is alleviated. To take the software engineering a step further, the data structures are designed to be as general as possible. Thus, it is an easy task to modify or rectify these structures to meet the demands of an evolving system.

A final characteristic of a sound Daplex interface is extensibility. A software product has to be designed in a manner that permits the easy modification and addition of code. In this light, we have placed "stubs" in appropriate locations within KFS to permit easy insertion of the code needed to handle multiple horizontal screens of output.

C. THE DATA STRUCTURES

The Daplex language interface has been developed as a single-user system. It is recognized however, that at some point in time the Daplex interface will be updated to a multi-user system. Accordingly, two different concepts of data are used in the interface: (1) data structures shared by all users, and (2) structures specific to each user. In accordance with the first data structure concept, the Daplex implementation has, whenever possible, used and added to the existing generic data structures generated by the previous implementations of DL/I and SQL. However, due to the complexity of the entity-relationship model, an additional large set of unique and specific data structures was required for the Daplex implementation.

1. Data Shared by All Users

The following discussion of data structures makes extensive use of the university database described in Figure 3. A frequent reference to Figure 3 may aid the reader greatly in understanding the following material.

The data structures that are shared by all users are the database schemas defined by the users thus far. In our case, these are entity-relationship schemas, consisting of entities and the relationships (functions) between the entities. These are not only shared by all users, but also shared by the four modules of the MLDS, i.e., LIL, KMS, KC, and KFS. It is important to note that this structure is represented as a union and is generic in the sense that it can be used to support the SQL, CODASYL, DL/I, and Daplex needs. Figure 4 depicts this data structure.

```
union  dbid_node
 {
   struct  rel_dbid_node  *rel;
   struct  hie_dbid_node  *hie;
   struct  net_dbid_node  *net;
   struct  ent_dbid_node  *ent;
 }
```

Figure 4. The dbid_node Data Structure.

The main concern of this thesis, however, is with the entity-relationship model. In this regard, the fourth field of this structure points to a node that contains the information about an entity-relationship database. Figure 5 illustrates this record.

The first field is simply a character array containing the name of the entity-relationship database. The second field contains a pointer to the base-type nonentity node, and the following field simply contains an integer value that represents the number of these nodes in the database. The fourth field points to the entity node, and as before the field that immediately follows contains an integer value representing the number of such nodes. The sixth field contains a pointer to the generalized entity supertype node and the seventh field the integer value of the number of these supertypes. The eighth and tenth fields contain

```
struct ent_dbid_node
    {
    char      edn_name[DBNLength + 1];
    struct    ent_non_node    *edn_nonentity;
    int       edn_num_nonent;
    struct    ent_node        *edn_entity;
    int       edn_num_ent;
    struct    gen_sub_node    *edn_subptr;
    int       edn_num_gen;
    struct    sub_non_node    *edn_nonsubptr;
    int       edn_num_nonsub;
    struct    der_non_node    *edn_nonderptr;
    int       edn_num_der;
    struct    ent_dbid_node   *edn_next_db;
    };
```

Figure 5. The ent_dbid_node Data Structure.

pointers to the nonentity subtypes and nonentity derived types respectively, and the ninth and eleventh fields contain the integer value for the number of such nodes. Finally, the twelfth field points to the next entity-relationship database node.

Figure 6 depicts the entity node structure. The first field of this structure is a character array which holds the name of the entity, and the second field is an integer representation of the number of functions associated with the entity that this node represents. For instance, the "person" entity has two functions associated with it, "name" and "ssn". The third field is an integer representation

```
struct ent_node
    {
    char      en_name[ENLength + 1];
    int       en_num_funct;
    int       en_terminal;
    struct    function_node  *en_ftnptr;
    struct    ent_node       *en_next_ent;
    };
```

Figure 6. The ent_node Data Structure.

of a boolean function and indicates whether or not the entity is a terminal type, i.e., not a supertype.

The structure of the gen_sub_node is shown in Figure 7. The first field, similar to previous nodes, holds the name of the generalized entity subtypes. An example applied to the university data base is "support_staff". The second field holds the number of functions associated with each entity subtype, and the third field is an integer representation of a boolean function and holds a "1" if the generalized entity is a subtype and not a supertype.

```
struct gen_sub_node
      {
       char     gsn_name[ENLength + 1];
       int      gsn_num_funct;
       int      gsn_terminal;
       struct   overlap_ent_node  *gsn_entptr;
       int      gsn_num_ent;
       struct   function_node    *gsn_ftnptr;
       struct   overlap_sub_node  *gsn_subptr;
       int      gsn_num_sub;
       struct   gen_sub_node     *gsn_next_genptr;
      };
```

Figure 7. The gen_sub_node Data Structure.

The fourth field holds a pointer to the entity supertype. In the case of "employee" the supertype is "person". The fifth field indicates the number of those entities. The sixth field holds a pointer to a function associated with the generalized subtype, for instance, "salary". The seventh field holds a pointer to the subtype supertype. For example, the supertype for the subtype "support_staff" is "employee". The eighth field maintains a record of the number of such subtype supertypes. The final field simply points to the next gen_sub_node.

The ent_non_node record is shown in Figure 8, and contains information about each nonentity base-type in the database. The first field of the record holds the name of the nonentity node, for example, "rank_name". The second field holds the character that indicates the type of nonentity node, either "i",

integer; "e", enumeration; "f", floating point; "s", string; "b", boolean. The next field contains an integer that indicates the maximum length of the base-type value.

The fourth field contains an integer representation of a boolean value, a "1" or "0", that indicates whether or not there is a range associated with the nonentity node. For example, the nonentity "grade_point" has a range of 0.0 to 4.0, while "rank-name" is without a range. The fifth field contains an integer that represents the number of different values that the nonentity can assume. As an example, both "rank_name" and "semester_name" can assume three values, but "grade_point" can assume 40 different values.

```
struct ent_non_node
    {
     char      enn_name[ENLength + 1];
     char      enn_type;
     int       enn_total_length;
     int       enn_range;
     int       enn_num_values;
     struct    ent_value      *enn_value;
     int       enn_constant;
     struct    ent_non_node   *enn_next_node;
    };
```

Figure 8. The ent_non_node Data Structure

The sixth field contains a pointer to the actual value of the base-type, and the seventh field contains an integer representation of a boolean value that indicates if the actual value of the base-type is a constant. There are no constants in the university database, but, as an example, the value of the base-type could assume the constant value of pi (3.14159265) or Avogadro's number (6.023 X 10^23). The eighth and final field contains a pointer to the next nonentity node.

The sub_non_node is shown in Figure 9. This structure is almost identical in form and similar in purpose to the ent_non_node of Figure 6. The main difference in purpose between the two structures is that the ent_non_node is for a base-type nonentity and the sub_non_node is for a subtype nonentity. The difference in form between the two structures is the absence of constants in the

```
struct sub_non_node
     {
      char      snn_name[ENLength + 1];
      char      snn_type;
      int       snn_total_length;
      int       snn_range;
      int       snn_num_values;
      struct    ent_value       *snn_value;
      struct    sub_non_node    *snn_next_node;
     };
```

Figure 9. The sub_non_node Data Structure.

sub_non_node. Maintaining two separate constant lists would be redundant, hence the constants are found only in the ent_non_node.

The next node, similar to both the ent_non_node and the sub_non_node, is the der_non_node, shown in Figure 10. The der_non_node is identical in structure to the sub_non_node and differs in function in that it applies to the derived nonentity subtypes.

```
struct der_non_node
     {
      char      dnn_name[ENLength + 1];
      char      dnn_type;
      int       dnn_total_length;
      int       dnn_range;
      int       dnn_num_values;
      struct    ent_value       *dnn_value;
      struct    der_non_node    *dnn_next_node;
     };
```

Figure 10. The der_non_node Data Structure.

The final node that we will discuss is this section is the function_node, shown in Figure 11. The function_node defines the structures for each function type declaration. As an example, the function "dept" returns the entity "department" when applied to the entity "faculty".

The first field of the function_node points to the name of the function, in this example the name is "dept". The second field holds the type, an "e" in this

```
struct function_node
     {
      char     fn_name[ENLength+1];
      char     fn_type;
      int      fn_range;
      int      fn_total_length;
      int      fn_num_value;
      struct   ent_value    *fn_value;
      struct   ent_node     *fn_entptr;
      struct   gen_sub_node *fn_subptr;
      struct   ent_non_node *fn_nonentptr;
      struct   sub_non_node *fn_nonsubptr;
      struct   der_non_node *fn_nonderptr;
      int      fn_entnull;
      int      fn_unique;
      struct   function_node *fn_next_fntptr;
     };
```

Figure 11. The function_node Data Structure.

case. The third field indicates when the function has a range of values, the fourth field indicates the length and the fifth field indicates the number of values, if any. In this example, all three fields would hold a "0".

The sixth field would hold the actual value, if there were any, and the next five fields hold pointers to the type to which a particular function belongs. A function may belong to more than one type, but it is extremely unlikely that it would belong to all five. In our example, the function "dept" belongs to only one type, the entity "department", hence only the ent_node pointer, fn_entptr, will contain any information, the remaining four type field pointers will be empty.

The twelfth field indicates if there is an associated entity value. It is initialized to hold a "0" and in the above example maintains that "0". The thirteenth field indicates whether or not the function is unique. It too is initialized to "0", and in our example maintains that "0". The final field simply contains a pointer to the next function.

2. Data Specific to Each User

This category of data represents information required to support each user's particular interface needs. The data structures used to accomplish this

may be thought of as forming a hierarchy. At the root of this hierarchy is the user_info record, shown in Figure 12, which maintains information on all current users of a particular language interface. The user_info record holds the ID of the user, a union that describes a particular interface, and a pointer to the next user. The union field is of particular interest to us. As noted earlier, a union serves as a generic data structure.

```
struct  user_info
 {
  char              uid[UIDLength + 1];
  union   li_info    li_type;
  struct  user_info  *next_user;
 }
```

Figure 12. The user_info Data Structure.

In this case, the union may hold the data for a user accessing either an SQL language interface layer, a DL/I LIL, a CODASYL-DML LIL, or a Daplex LIL. The li_info union is shown in Figure 13.

We are only interested in the data structures containing user information that pertain to Daplex, or entity-relationship, language interface. This structure is referred to as dap_info and is depicted in Figure 14. The first field of this structure, dpi_curr_db, is itself a record and contains currency information on the database being accessed by a user. The second field, dpi_file, is also a record. The file record contains the file descriptor and file identifier of a file of Daplex transactions, either requests or database descriptions. The next field, dpi_dml_tran, is also a record, and holds information that describes the Daplex

```
union  li_info
   {
     struct  sql_info  li_sql;
     struct  dli_info  li_dli;
     struct  dml_info  li_dml;
     struct  dap_info  li_dap;
   }
```

Figure 13. The li_info Data Structure.

```
struct dap_info
    {
     struct    curr_db_info    dpi_curr_db;
     struct    file_info       dpi_file;
     struct    tran_info       dpi_dml_tran;
     int       dap_operation;
     struct    ddl_info        *dpi_ddl_files;
     union     kms_info        dpi_kms_data;
     union     kfs_info        dpi_kfs_data;
     union     kc_info         dpi_kc_data;
     int       dap_error;
     int       dap_answer;
     int       dap_buff_count;
    };
```

Figure 14. The dap_info Data Structure.

transactions to be processed. This includes the number of requests to be processed, the first request to be processed, and the current request being processed. The fourth field of the dap_info record, dap_operation, is a flag that indicates the operation to be performed. This may be either the loading of a new database, or the execution of a request against an existing database. The next field, dpi_ddl_files, is a pointer to a record describing the descriptor and template files. These files contain information about the ABDL schema corresponding to the current entity-relationship database being processed, i.e., the ABDL schema information for a newly defined entity-relationship database. The following fields, dpi_kms_data, dpi_kfs_data and dpi_kc_data, are unions that contain information required by the KMS, KFS and KC, respectively. These are described in more detail in later chapters. The next field, error, is an integer value representing a specific error type. The next field, answer, is used by the LIL to record answers received through its interaction with the user of the interface. The last field, buff_count, is a counter variable used in the KC to keep track of the result buffers.

D. THE ORGANIZATION OF THE NEXT FOUR CHAPTERS

The following four chapters are meant to provide the user with a more detailed analysis of the modules constituting MLDS and Daplex implementations. Each chapter begins with an overview of what each particular module does and how it relates to the other modules. The actual processes performed by each module are then discussed. This includes a description of the actual data structures used by the modules. Each chapter concludes with a discussion of module shortcomings.

III. STORAGE AND RETRIEVAL OF THE DAPLEX SCHEMAS

The first modules that we discuss concern the storage of the Daplex schemas from memory and the recreation of those schemas in memory from a file. It is understood that these modules are not as conceptually interesting as the LIL, KMS, KC or KFS, but they are important, and are included here for completeness.

The reader is reminded that Appendix B contains the modules for storage and retrieval and should be consulted frequently to ensure a thorough understanding of this chapter.

A. DAPLEX SCHEMA STORAGE

Early in the design phase of the storage module, we realized that several items in the schema could be stored more than once and storage space unnecessarily wasted. Accordingly, we made a concerted effort to avoid storing redundant data and mapped the data to the correct structure with the use of pointers.

The Daplex schemas are tied together as they are written to a file by a series of pointer manipulations. The pointer that is responsible for each ent_dbid_node, hence for the entire Daplex database, is known as db_ptr. Generally, the db_ptr is set to the head of the database and then passed to the routine responsible for writing the contents of that specific portion of the ent_dbid_node to a file. Accordingly, the entire ent_dbid_node is not written to the file at this time, rather, only the database name, edn_name, and the number of nonentities, edn_num_nonent, are stored. (see Figure 5 again) In general, as the pointer is sequenced through the node, each structure it encounters is processed in turn, storing necessary information while at the same time avoiding information that may be previously stored in another node.

The first structure that the pointer encounters is the ent_non_node (as in Figure 5). The routine for storing the nonentity nodes is known as proc_ent_non_node. The entire nonentity node is stored at this time, (see Figure 8 again) including any associated entity values, as this information is not

duplicated elsewhere. The pointer in the calling routine then moves to the next nonentity node and the entire procedure is repeated. This process continues until all the nonentity nodes have been stored.

The db_ptr, is then set to the entity nodes (as in Figure 5), the procedure for processing the entity nodes, wr_ent_node, is executed, and the entire set of entity nodes is processed. (see Figure 6 again) However, the functions associated with the respective entities are not processed, as all the functions are handled separately.

The routine wr_gen_sub_node, processes the next node, the generalized entity subtypes. Only the first three fields, gsn_name, gsn_num_funct, and gsn_terminal, (see Figure 7 again) are stored directly by this routine, the remaining fields are stored immediately after execution of wr_gen_sub_node, and within the main routine.

The remainder of the gen_sub_node is processed in the main routine. This segment of code is handled in the main routine instead of in a separate procedure because the pointer manipulations are more easily handled here and because this data is processed only once, a separate routine was not considered necessary. The reader should note that the subtypes with entity supertypes, i.e., the overlap_ent_nodes, and the terminal subtypes that define one or more subtypes, i.e., the overlap_sub_nodes, that are associated with the gen_sub_node are all processed at this time.

The subtype nonentity nodes are processed within the proc_sub_non_node routine. First, the db_ptr is set to point at the edn_nonsubptr (as in Figure 5). The proc_sub_non_node is then called, and the entire sub_non_node is stored.

The derived type nonentity nodes are processed in exactly the same manner as the sub_non_node. The db_ptr is set to point at edn_nonderptr (as in Figure 5) and the entire der_non_node is processed. (see Figure 10 again)

The functions associated with the entity nodes are the next items stored. The reader may remember that we chose not to store these functions earlier. We store the functions now by first setting the db_ptr to point at the entity nodes and then call wr_all_ent_node, a routine that calls a second routine, proc_function_node, that processes all of the functions.

The proc_function_node routine tracks sequentially through the appropriate function node (see Figure 11 again) and stores data for every field. Since the ent_node, gen_sub_node, ent_non_node, sub_non_node and der_non_node (as in Figure 5) may or may not have data aasociated with them, we have chosen to place a "^" in the empty fields to maintain the integrity of the database.

The functions associated with the gen_sub_nodes (as in Figure 7) are stored in a manner very similar to the process just described. The db_ptr is set to point at the gen_sub_node and once again the proc_function_node routine is called.

At this point the database for one ent_dbid_node has been stored. The program checks to see if any more ent_dbid_nodes remain to be stored. If so, the above procedures are repeated. If not, a "$" is inserted at the end of the database as an end of database marker.

B. RETRIEVAL OF THE DAPLEX SCHEMA

The process for retrieving the Daplex schemas from secondary storage and loading them into memory is almost a reverse of the storage procedure. Different structures are used, as the save module was written by one member of the team and the retrieval module by the other; otherwise, the process is basically a reversal.

The routine that reads data into the first Daplex database, i.e., the first ent_dbid_node, is rd_ent_dbid_node. The memory is first allocated, the pointers nulled, and then the first two fields, edn_name and edn_num_nonent, (as in Figure 5) are loaded into memory. The remaining fields are loaded in order along with the respective field data.

The pointer sequences to the next allocated space in memory and the routine that reads in the data for the nonentity nodes, rd_ent_non_node, is executed (as in Figure 8). As before, the entire nonentity node is processed at one time.

The next structures to be filled, at least partially, are the entity nodes (as in Figure 6). As with wr_ent_node, the functions associated with rd_ent_node, and therefore the entities, are processed later.

The generalized entity subtypes are the next nodes to be processed. As with the storage routine, only three of the fields gsn_name, gsn_num_funct and gsn_terminal, are processed in rd_gen_sub_node (as in Figure 7). However,

unlike the storage routine, the remainder of the gen_sub_node is handled in two smaller routines, rd_overlap_ent_node and rd_overlap_sub_node.

After the first three fields of the generalized subtypes are processed, the pointer is sequenced and the routine that handles the subtypes with one more entity supertypes, rd_overlap_ent_node, is executed. The rd_overlap_ent_node routine checks for the presence of subtypes with one or more supertypes and then loads those names into memory. The routine rd_overlap_sub_node functions exactly as rd_overlap_ent_node, but on the subtype supertypes.

After the overlap nodes are processed, the pointer sequences and the subtype nonentity nodes are allocated and filled. This process occurs within the rd_sub_non_node routine (as in Figure 9).

The derived type nonentity nodes (as in Figure 10) are processed in exactly the same manner as the sub_non_node. The pointer is sequenced, the memory allocated, and data entered in exactly the same fashion. The functions associated with the entity nodes are the next items loaded into memory. The functions are loaded by first sequencing the pointer and then calling the routine responsible for loading the functions, rd_function_node. The rd_function_node routine, along with the previous routines, first allocates the necessary memory and then nulls the appropriate pointers. The routine then tracks through the function node (as in Figure 11) and loads those fields with data. Since it is possible for any of the ent_nodes, gen_sub_nodes, ent_non_nodes, sub_non_nodes or der_non_nodes to be without data, the routine first checks those nodes to see if they contain a "^", the symbol for an empty node. Finally, the module checks to see if it has encountered a "$", the symbol for end of database. If so, all processes are terminated.

We have written a small main routine that first executes the retrieval of an existing database and then executes the saving of that database to a file. The main routine calls the two modules previously discussed and then executes a print statement for every retrieval and save action. This methodology has allowed the authors to more effectively debug the programs.

IV. THE LANGUAGE INTERFACE LAYER (LIL)

The second set of modules that we will discuss concern LIL, the first modules in the Daplex mapping process. LIL is used to control the order in which the other modules are called, and allows the user to input transactions from either a file or the terminal. A transaction may take the form of either a database description (DBD) of a new database, or a Daplex request against an existing database. A single transaction may contain multiple requests, allowing a group of requests to perform a single task. For example, several "atomic" statements, those statements that are executed as an indivisible action with respect to the database, could be executed together as a single transaction.

The mapping process occurs when LIL sends a single transaction to KMS. After the transaction has been received by KMS, KC is called to process the transaction. Control always returns to LIL, where the user may either continue with another transaction or close the session by exiting to the operating system.

LIL is menu-driven, and when the transactions are read from either a file or the terminal, they are stored in the dap_req_info data structure. If the transactions are database descriptions, they are sent to the KMS in sequential order. If the transactions are Daplex requests, the user is prompted by another menu to selectively choose an individual request to be processed. The menus provide an easy and efficient way for the user to view and select the methods of request processing desired. Each menu is tied to its predecessor, so that by exiting one menu the user is moved up the "menu tree". This allows the user to perform multiple tasks in one session.

A. THE LIL DATA STRUCTURES

LIL uses three data structures to store the user's transactions and control the transaction sent to KMS. It is important to note that these data structures are shared by both LIL and KMS.

The first data structure is named tran_info and is shown in Figure 15. The first field of this record, ti_first_req, is the pointer to the first request data

```
struct tran_info
      {
        union    req_info    ti_first_req;
        union         req_info    ti_curr_req;
        int      ti_no_req;
      };
```

Figure 15. The tran_info Data Structure.

structure that contains the union of all the language requests of MLDS (see Figure 16). The first request can originate from either a file or a terminal. The second field of tran_info is a pointer to the current transaction, set by LIL to tell the KMS the precise transaction to process next. The third field contains the number of transactions currently in the transaction list. This number is used for loop control when printing the transaction list to the screen, or when searching the list for a transaction to be executed.

The second data structure used by LIL, req_info, is a union of the language requests of MLDS, and is shown in Figure 16. It serves a routing control function, in that it routes a transaction request to the appropriate database language. In this thesis, we are concerned only with the the fourth field of this structure, which contains a pointer to the dap_req_info data structure (see Figure 17), each copy representing a Daplex user transaction.

The third data structure used by LIL is named dap_req_info. Each copy of this record represents a user transaction, and thus, is an element of the transaction list. The dap_req_info data structure is shown in Figure 17. The first field

```
union req_info
      {
        struct    rel_req_info    *ri_rel_req;
        struct    hie_req_info    *ri_hie_req;
        struct    net_req_info    *ri_net_req;
        struct    dap_req_info    *ri_dap_req;
        struct    ab_req_info     *ri_ab_req;
      };
```

Figure 16. The req_info Data Structure.

```
struct  dap_req_info
        {
         char     *dap_req;
         int      dap_req_len;
         struct   temp_str_info    *dap_in_req;
         struct   dap_req_info     *dap_sub_req;
         struct   dap_req_info     *dap_next_req;
        };
```

Figure 17. The dap_req_info Data Structure.

of this record, dap_req, is a character string that contains the actual Daplex transaction. The second field, dap_req_len, contains the length of the transaction. It is used to allocate the exact, and therefore minimal, amount of memory space for the transaction. The third field, dap_in_req, is a pointer to a list of character arrays that each contain a single line of one transaction. After all lines of a transaction have been read, the line list is concatenated to form the actual transaction, dap_req. If a transaction contains multiple requests, the fourth field, dap_sub_req, points to the list of requests that make up the transaction. In this case, the field dap_in_req is the first request of the transaction. The last field, dap_next_req, is a pointer to the next transaction in the list of transactions.

B. FUNCTIONS AND PROCEDURES

LIL makes use of a number of functions and procedures in order to create the transaction list, pass elements of the list to KMS, and maintain the database schemas. We do not describe each of these functions and procedures in detail. Rather, we provide a general description of the LIL processes.

1. Initialization

The MLDS is designed to be able to accommodate multiple users, but in this version it is implemented to support only a single user. To facilitate the transition from a single-user system to a multiple-user system, each user possesses his own copy of a user data structure when entering the system. This user data structure stores all of the relevant data that the user may need during their session. All four modules of the language interface make use of this structure. The modules use many temporary storage variables, both to perform their individual

tasks, and to maintain common data between modules. The transactions, in user data language form, and mapped kernel data language form, are also stored in each user data structure. It is easy to see that the user structure provides consolidated, centralized control for each user of the system. When a user logs onto the system, a user data structure is allocated and initialized. The user ID becomes the distinguishing feature to locate and identify different users. The user data structures for all users are stored in a linked-list. When new users enter the system, their user data structures are appended to the end of the list. In our current environment there is only a single element on the user list. In a future environment, when there are multiple users, we simply expand the user list as described above.

2. Creating the Transaction List

There are two operations the user may perform. A user may define a new database or process Daplex requests against an existing database. The first menu that is displayed prompts the user to select the operation desired. Each operation represents a separate procedure to handle specific circumstances. The menu looks like the following:

```
Enter type of operation desired
     (l) - load a new database
     (p) - process old database
     (x) - return to the operating system
ACTION ----> _
```

For either choice (i.e., l or p), another menu is displayed to the user requesting the mode of input. This input may always come from a data file. If the operation selected from the previous menu had been "p", then the user may also input transactions interactively from the terminal. The generic menu looks like the following:

Enter mode of input desired
(f) - read in a group of transactions from a file
(t) - read in transactions from the terminal
(x) - return to the previous menu
ACTION ----> _

Note that the choice "t" would be omitted if the operation selected from the previous menu had been to load a new database. Again, each mode of input selected corresponds to a different procedure to be performed. The transaction list is created by reading from the file or terminal, looking for an end-of-transaction marker or an end-of-file marker. These flags tell the system when one transaction has ended, and when the next transaction begins. When the list is being created, the pointers to access the list are initialized. These pointers, ti_first_req and ti_curr_req (as in Figure 15) are set to the first transaction read, in other words, to the head of the transaction list.

3. Accessing the Transaction List

Since the transaction list stores both DBDs and Daplex requests, two different access methods have to be employed to send the two types of transactions to the KMS. We discuss the two methods separately. In both cases, the KMS accesses a single transaction from the transaction list. It does this by reading the transaction pointed to by the request pointer, ti_curr_req, of the tran_info data structure (as in Figure 15). Therefore, it is the job of LIL to set this pointer to the appropriate transaction before calling KMS.

a. Sending DBDs to KMS - When the user specifies the filename of DBDs (input from a file only), further user intervention is not required. To produce a new database, the transaction list of DBDs is sent to KMS via a program loop. This loop traverses the transaction list, calling KMS for each DBD in the list.

b. Sending Daplex Requests to KMS - In this case, after the user has specified the mode of input, the user conducts an interactive session with the system. First, all Daplex requests are listed to the screen. As the requests are listed from the transaction list, a number is assigned to

each transaction in ascending order, starting with the number one. The number appears on the screen to the left of the first line of each transaction. Note that each transaction may contain multiple requests. Next, an access menu is displayed which looks like the following:

```
Pick the number or letter of the action desired
     (num) - execute one of the preceding transactions
     (d)   - redisplay the list of transactions
     (r)   - reset the currency pointer to the root
     (x)   - return to the previous menu
ACTION ----> _
```

Since Daplex requests are independent items, the order in which they are processed does not matter. The user has the option of executing any number of Daplex requests. A loop causes the menu to be redisplayed after any Daplex request has been executed so that further choices may be made. The selection "r" causes the currency pointer to be repositioned to the root of the entity-relationship schema so that subsequent requests may access the complete database, rather than be limited to beginning from a current position established by previous requests.

4. Calling the KC

As mentioned earlier, LIL acts as the control module for the entire system. When KMS has completed its mapping process, the transformed transactions have to be sent to KC to interface with the kernel database system. For DBDs, KC is called after all DBDs on the transaction list have been sent to KMS. The mapped DBDs have been placed in a mapped transaction list that KC is going to access. Since Daplex requests are independent items, the user should wait for the results from one Daplex request before issuing another. Therefore, after each Daplex request has been sent to KMS, KC is immediately called. The mapped Daplex requests are placed on a mapped transaction list, which KC may easily access.

5. Wrapping-up

Before exiting the system, the user data structure described in Chapter II (as in Figure 12) has to be deallocated. The memory occupied by the user data

structure is freed and returned to the operating system. Since all of the user structures reside in a list, the exiting user's node has to be removed from the list.

V. THE KERNEL MAPPING SYSTEM (KMS)

KMS is the second module in the Daplex mapping interface and is called from the language interface layer (LIL) when LIL has received Daplex requests input by the user. The function of KMS is to: (1) parse the request to validate the user's Daplex syntax, (2) translate, or map, the request to an equivalent ABDL request, and (3) perform a semantic analysis of the current ABDL request generated relative to the request generated during a previous call to KMS. Once an appropriate ABDL request, or set of requests, has been formed, it is made available to the kernel controller (KC) which then prepares the request for execution by MBDS. KC is discussed in Chapter VI.

A. AN OVERVIEW OF THE MAPPING PROCESS

From the description of the KMS functions above we immediately see the requirement for a parser as a part of the KMS. This parser validates the Daplex syntax of the input request. The parser grammar is the driving force behind the entire mapping system.

1. The KMS Parser / Translator

The KMS parser has been constructed by utilizing Yet-Another-Compiler Compiler (YACC) [Ref. 17]. YACC is a program generator designed for syntactic processing of token input streams. Given a specification of the input language structure (a set of grammar rules), the user's code to be invoked when such structures are recognized, and a low-level input routine, YACC generates a program that syntactically recognizes the input language and allows invocation of the user's code throughout this recognition process. The class of specifications accepted is a very general one: LALR(1) grammars. It is important to note that the user's code mentioned above is our mapping code that is going to perform the Daplex-to-ABDL translation. As the low-level input routine, we utilize a Lexical Analyzer Generator (LEX) [Ref. 18]. LEX is a program generator designed for lexical processing of character input streams. Given a regular-expression

description of the input strings, LEX generates a program that partitions the input stream into tokens and communicates these tokens to the parser.

The parser produced by YACC consists of a finite-state automaton with a stack that performs a top-down parse, with left-to-right scan and a one token look-ahead. Control of the parser begins initially with the highest-level grammar rule. Control descends through the grammar hierarchy, calling lower and lower-level grammar rules while searching for appropriate tokens in the input. As the appropriate tokens are recognized, some portions of the mapping code are invoked directly. In other cases, tokens are propagated upwards through the grammar hierarchy until a higher-level rule has been satisfied, and a further translation is performed. When all of the necessary lower-level grammar rules have been satisfied and control has ascended to the highest-level rule, the parsing and translation processes are complete. In Section B, we give an illustrative example of these processes. We also describe the subsequent semantic analysis necessary to complete the mapping process. The reader is reminded that Appendix C contains the code for our implementation, written in C.

2. <u>The KMS Data Structures</u>

KMS utilizes just two structures that are defined in the interface. Naturally, KMS requires access to the Daplex input request structure discussed in Chapter II, the dpi_dml_tran (see Figure 14 again) structure. However, the only two data structures to be discussed here are those unique to the KMS.

Both of these structures are shown in Figure 18. The first of these, dap_kms_info, is a record that contains information, not of immediate use, that has been accumulated by the KMS during the grammar-driven parse. This record allows the information to be saved until a point in the parsing process where it may be utilized in the appropriate portion of the translation process. The first four fields in this record, point to the same structure, ident_list, the second structure of Figure 18, which temporarily holds a list of names for comparison with the identifiers, subtype indicators, overlap_sub_node or overlap_ent_node, and uniqueness identifiers, respectively. These names are those of attributes whose values are retrieved from the database. The remaining fields of dap_kms_info contain pointers to Daplex node structures previously discussed

```
struct  dap_kms_info
 {
  struct  ident_list     *dki_temp_ptr;
  struct  ident_list     *dki_id_ptr;
  struct  ident_list     *dki_overfirst_ptr;
  struct  ident_list     *dki_name1_ptr;
  struct  der_non_node    dki_der_non;
  struct  sub_non_node    dki_sub_non;
  struct  ent_non_node    dki_ent_non;
  struct  function_node   dki_funct;
  struct  ent_value       *dki_ev_ptr;
 };

struct  ident_list
 {
  char   il_name [ENlength + 1];
  struct ident_list       *il_next;
 };
```

Figure 18. The KMS Data Structures.

in Chapter II. The remaining field of ident_list points to the next name in the list. At the conclusion of the mapping process, and before control is returned to LIL, all data structures unique to KMS that have been allocated during the mapping process are freed.

B. POSSIBLE FACILITIES PROVIDED BY AN IMPLEMENTATION

As we reached this stage in the implementation, we were confronted with two problems. First, the deadline date for completion of this project was rapidly approaching, and second, the amount of code left to produce was nearly equal to the amount of code that we had provided to this point. In addition, due to the complexity of the entity-relationship model, the amount of code produced for the Daplex implementation had met or exceeded the amount of code for each of the implementations of DL/I, SQL, and CODASYL [Refs. 12, 13 and 19]. Accordingly, a decision was made to discontinue the implementation effort for this thesis and leave the remainder for another thesis.

In the remainder of this chapter, we discuss those Daplex facilities that may be provided by an implementation of the entity-relationship interface. We do not discuss the Daplex-to-ABDL translation in detail. Rather, we provide only an overview of the salient features of KMS. The interested reader is referred to Goisman [Ref. 11], for a detailed discussion of the Daplex-to-ABDL translation. User-issued requests may take two forms, either Daplex database definitions, or Daplex database manipulations. In the case of database manipulations, we also describe the semantic analysis necessary to complete the mapping process.

1. Database Definitions

When the user informs the LIL that the user wishes to create a new database, the job of the KMS is to build a entity-relationship database schema that corresponds to the database definition input by the user. The LIL initially allocates a new database identification node (ent_dbid_node shown in Figure 5) with the name of the new database, as input by the user. The LIL then sends the KMS a complete database description which takes the form of a Daplex database declaration as follows:

```
DATABASE db_name IS
  [non_entity_type_declarations]
  entity_type_declarations
  [entity_type_constraints]
END [db_name];
```

Where:

db_name: is a valid identifier that is a unique
name of the database being declared.

non_entity_type_declarations: are declarations
of string types, scalar types, and
numeric constants.

entity_type_declarations: are declarations of
entity types, their functions, and
generalization hierarchies.

entity_type_constraints: define those properties
of the declared entity type that must
remain invariant under any operation on
values of those types.

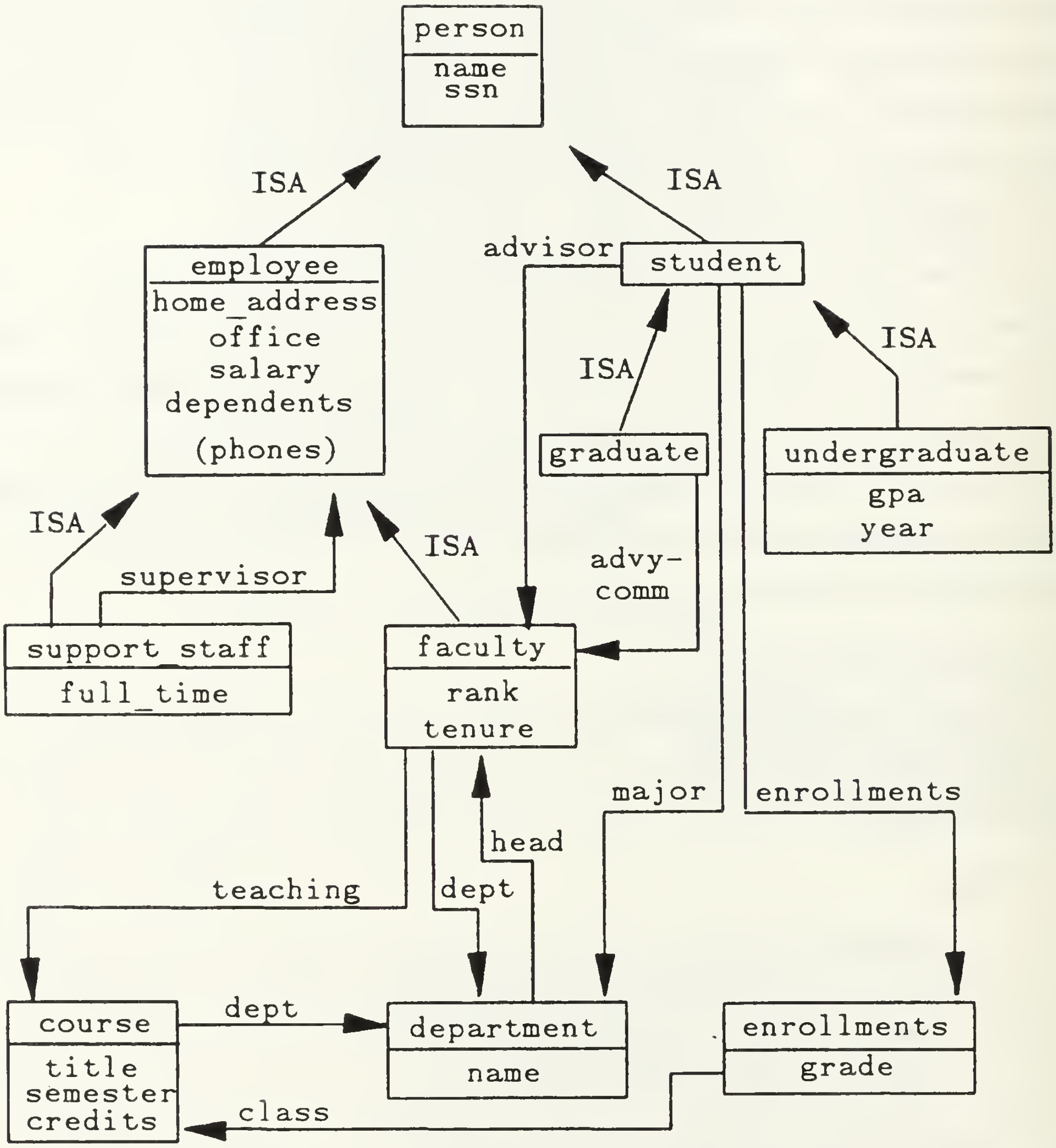


Figure 19. The University Database Schema

The non_entity_type_declarations, entity_type_declarations, and entity_type_constraints that form a database declaration can be intermixed in any order. However, all types must be declared (either completely or partially) before the name of the type can appear in another declaration. Accordingly, it is apparent that for each ent_dbid_node, a differing mix of ent_node(s), gen_sub_node(s), ent_non_node(s) and function_node(s) is possible.

When LIL has forwarded all database definitions entered by the user, a completed database schema is the result. A completed database schema that uses the University database of Figure 3 is shown in Figure 19. The entity-relationship database schema, when completed, serves two purposes. First, when creating a new database, it facilitates the construction of the MBDS template and descriptor files. Secondly, when processing requests against an existing database, it allows a validity check of the entity, nonentity, and function names. It also serves as a source of information for the type checking.

2. Database Manipulations

When the user wishes LIL to process requests against an existing database, the first task of the KMS is to map the user's Daplex request to an equivalent ABDL request. The only ABDL requests available are RETRIEVE, RETRIEVE-COMMON, INSERT, UPDATE and DELETE. To these ABDL requests KMS must map the Daplex operators ASSIGNMENT, INCLUDE, EXCLUDE, CREATE, DESTROY, MOVE and PROCEDURE_CALL.

We will not discuss PROCEDURE_CALL as it includes utility procedures such as print and cancel, and these operations are accommodated by the MLDS and ABDL operators. In addition, we will not discuss the RETRIEVE-COMMON statement of ABDL. Further, the mappings will be discussed at a level of abstraction that does not imply a specific coding implementation, but rather, a general algorithm that will accomplish the mapping.

The first mapping that we will discuss is the CREATE mapping. A CREATE statement is used to create a new database entity. The structure for CREATE is shown in Figure 20.

The function names and values are those function pairs that are associated with a specific entity type or entity subtype. The entity types and entity

```
CREATE

        list of function names
        list of function values
        list of entity types and entity subtypes
              to be created
        pointer to RETRIEVE
        pointer to INSERT
              or
        pointer to INSERT
              .
              .
              .
        pointer to INSERT
```

Figure 20. The CREATE Data Structure.

subtypes to be used to CREATE a new database entity are maintained in a list and the creation process continues as long as there are entity types or entity subtypes in the list.

In general, the CREATE algorithm first determines if the new entity associated with the function pair in question is an existing supertype or a terminal type. If so, then the appropriate supertype/terminal type previously associated with the function pair is RETRIEVEd from the database, and the new entity type or entity subtype is INSERTed. Otherwise, the new entity type or entity subtype is simply INSERTed into the database.

The next mapping that we will discuss is the DESTROY mapping shown in Figure 21. The function names and values for the DESTROY structure are the same as those associated with the CREATE structure, and in fact, these function pairs are the same for all of the subsequent Daplex mappings that we will discuss. The entity types and entity subtypes that are to be DESTROYed are maintained in a list and the destruction process continues as long as there are items in the list to be DESTROYed.

The DESTROY algorithm only DELETEs entities, and further, only DESTROYs those entities that are not referenced by some database function. Therefore an entity is RETRIEVEd and a determination made as to whether the

DESTROY

list of function names
list of function values
list of entity types and entity subtypes
to be destroyed
pointer to RETRIEVE
pointer to DELETE
.
.
.
pointer to DELETE

Figure 21. The DESTROY Data Structure.

entity is referenced by a database function. If so, then the DESTROY operation is aborted. If not, the entity is DELETEd and the process continues for the next entity to be DESTROYed until the list is empty.

The FOR EACH structure is shown in Figure 22. The FOR EACH structure uses the set of database values as a pivot for the iteration process. Each element of the set of database values is paired with a set_expression for the execution of the loop. The set_expression values may be entites, function names or function values and provide the set of values over which the loop is iterated. Each RETRIEVE is accomplished on a set_expression value and an element of

FOR EACH

list of sets of database values
set_expression values
pointer to RETRIEVE
.
.
.
pointer to RETRIEVE

Figure 22. The FOR EACH Data Structure.

the set of database values acts as the second arguement for the operation to be carried out by FOR EACH. The RETRIEVEs continue until the list of sets of database values is empty.

The ASSIGNMENT statement structure is shown in Figure 23. The purpose of the ASSIGNMENT statement is to assign entity values to single-valued functions.

ASSIGNMENT

list of function names
list of function values
list of entity types and subtypes
of each function
pointer to RETRIEVE
or
pointer to RETRIEVE
pointer to RETRIEVE
pointer to RETRIEVE
.
.
.
pointer to RETRIEVE
REPEAT
pointer to RETRIEVE
or
pointer to RETRIEVE
pointer to RETRIEVE
pointer to RETRIEVE
pointer to RETRIEVE
.
.
.
pointer to RETRIEVE
pointer to RETRIEVE
pointer to RETRIEVE
pointer to UPDATE
.
.
.
pointer to UPDATE

Figure 23. The ASSIGNMENT Data Structure.

purpose of the ASSIGNMENT statement is to assign entity values to single-valued functions. To accomplish this, the ASSIGNMENT algorithm searches through the database by RETRIEVing and comparing the function to be assigned to all of the functions in the database. In this case, it is assumed that the function in question exists and can be found.

The search is accomplished by going first to a supertype and comparing the functions associated with each of the subtypes until a match is found. If no match is found, then each subtype is treated as a supertype and the search continues downward through the tree until the function is found or a terminal type is reached.

```
                  INCLUDE

         list of function names
        list of function values
        list of entity types and subtypes
                of each function
         pointer to RETRIEVE
                or
         pointer to RETRIEVE
         pointer to RETRIEVE
         pointer to RETRIEVE
                .
                .
                .
         pointer to RETRIEVE
        pointer to INSERT
                .
                .
                .
        pointer to INSERT
```

Figure 24. The INCLUDE Data Structure.

Once the function is found, the search begins for all of the function values. This searching process is similar to the process for the function lookup and repeats until the desired value associated with the function in question is found. Once found, the value is then UPDATEd. The entire process continues until the list of functions and values to be assigned is empty.

The INCLUDE statement structure is shown in Figure 24. The purpose of the INCLUDE statement is to add either a single value or a set of values to a set-valued function. It functions in a manner similar to the ASSIGNMENT statement in that the search is accomplished by going first to a supertype and then comparing the functions associated with each of the subtypes until a match with the desired function is found. If no match is found, then each subtype is treated as a supertype and the search continues downward through the tree until the function is found or a terminal type is reached. Once the function is found, the single value or set of values that the user wishes to INCLUDE is INSERTed.

EXCLUDE

list of function names
list of function values
list of entity types and subtypes
of each function
pointer to RETRIEVE
or
pointer to RETRIEVE
pointer to RETRIEVE
pointer to RETRIEVE
.
.
.
pointer to RETRIEVE
pointer to DELETE
.
.
.
pointer to DELETE

Figure 25. The EXCLUDE Data Structure.

The EXLUDE statement structure is almost identical to the INCLUDE statement structure. As can be seen from Figure 25, the only difference is that once the desired function is found the value is DELETEd instead of INSERTed.

The final structure that we will discuss is that of the MOVE statement, shown in Figure 26. The purpose of the MOVE statement is to change the subtypes to which an entity belongs. The MOVE statement algorithm first performs

MOVE

list of function names
list of function values
list of entity types and subtypes
to be moved
pointer to RETRIEVE
pointer to RETRIEVE
pointer to DELETE
pointer to RETRIEVE
pointer to INSERT
.
.
.
pointer to INSERT

Figure 26. The MOVE Data Structure.

a RETRIEVE from the database using functions in entity valued expressions as a search key, or just using the entity valued expressions if the associated functions are not given. When the entity valued expressions are located, the corresponding functions are then searched for and RETRIEVEd. The entity valued expression is then DELETEd from its current location in the database and the new entity to which the entity valued expression is to be associated is RETRIEVEd. The entity valued expression and its associated function is then INSERTed into the new location.

VI. CONCLUSION

In this thesis, we have presented a partial specification and implementation of a Daplex language interface . This is one of four language interfaces that the multi-lingual database system will support. When complete, the multi-lingual database system will be able to execute transactions written in four well-known and important data languages, namely, SQL, DL/I, Daplex, and CODASYL. In our case, we support Daplex transactions with our language interface by way of a LIL and KMS, and have left the production of a Daplex KC and KFS for a future thesis. Related theses by Benson and Wentz, Kloepping and Mack, and Emdi [Refs. 12, 13 and 19] have examined the specification and implementation of the DL/I, SQL and CODASYL-DML language interfaces, respectively. All of these works are a part of the ongoing research being conducted at the Laboratory for Database Systems Research, Naval Postgraduate School, Monterey, California.

The need to provide an alternative to the development of separate stand-alone database systems for specific data language models has been the motivation for this research. In this regard, we have first demonstrated the feasibility of a multi-lingual database system (MLDS) by showing how a software Daplex language interface can be constructed.

A major goal has been to design a Daplex-to-MBDS interface without requiring any change be made to MBDS or ABDL. Our partial implementation may be completely resident on a host computer or the controller. All Daplex transactions will be performed in the Daplex interface. MBDS continues to receive and process transactions written in the unaltered syntax of ABDL. In addition, our implementation has not required any change to the syntax of Daplex. The interface will be completely transparent to the Daplex user as well as to the MBDL.

In retrospect, our level-by-level, top-down approach to the design of the interface has been a good choice. This implementation methodology has been the

most familiar to us and proved to be relatively efficient in time. In addition, this approach permits follow-on programmers to easily maintain and modify (when necessary) the code. Subsequently, they will know exactly where we have stopped and where they should begin because we have included many of the lower-level stubs. Hence, it is an easy task to fill in these stubs with code.

To our great disappointment we have not been able to complete the implementation. The primary reason for our failure has been the complexity of the entity-relationship model and the Daplex language. This complexity has been directly responsible for our underestimation of the amount of code necessary for the Daplex interface implementation. To date, we have produced an amount of code at least equal to each of the other complete implementations, and are faced with producing an equal amount in order to complete the implementation.

However, we have shown that a Daplex interface can be implemented as part of a MLDS. We have provided a partial software structure to facilitate this interface, and we have developed actual code for implementation. The next step is to complete the development of the Daplex interface. When complete, this interface can be integrated with the other implementations and tested as a whole to determine how efficient, effective, and responsive it can be to a users' needs. The results may be the impetus for a new direction in database system research and development.

APPENDIX A

DAPLEX DATA STRUCTURES

/* this is a list of the data structures for the daplex project */

```
union   dbid_node
        /* Union definition for the database. There is a common */
        /* database node definition that spans the four types of */
        /* language interfaces. Abbr: rel(ational), hie(archical),*/
        /* net(work), and entity-relationship.   */
        {
          struct   rel_dbid_node   *dn_rel;
          struct   hie_dbid_node   *dn_hie;
          struct   net_dbid_node   *dn_net;
        struct   ent_dbid_node   *dn_dap;
        };


struct ent_dbid_node
   /* structure def for each entity-relationship dbid node */
     {
     char     edn_name[DBNLength + 1];
     struct   ent_non_node    *edn_nonentity;
     int      edn_num_nonent; /* number of nonentity types */
     struct   ent_node        *edn_entity;
     int      edn_num_ent;    /* number of entity types */
     struct   gen_sub_node    *edn_subptr;
     int      edn_num_gen;    /* number of gen_subtypes */
     struct   sub_non_node    *edn_nonsubptr;
     int      edn_num_nonsub; /* number of nonentity subtypes */
     struct   der_non_node    *edn_nonderptr;
     int      edn_num_der;    /* nmbr or nonentity derived types */
     struct   ent_dbid_node   *edn_next_db;
     };


struct ent_node
   /* structure definition for each entity node */
       {
       char     en_name[ENLength + 1];
       int      en_num_funct;  /* number of assoc. functions */
       int      en_terminal;   /* if true (=1) it is a terminal type */
       struct   function_node  *en_ftnptr;
       struct   ent_node       *en_next_ent;
       };
```

```
struct gen_sub_node
  /* structure def for each generalization (supertype/subtype) node */
      {
       char     gsn_name[ENLength + 1];
       int      gsn_num_funct; /* number of assoc. functions*/
       int      gsn_terminal;  /* if true (=1) it is a terminal type */
       struct   overlap_ent_node  *gsn_entptr; /* ptr to entity supertype */
       int      gsn_num_ent;   /* number of entity supertypes */
       struct   function_node     *gsn_ftnptr;
       struct   overlap_sub_node  *gsn_subptr; /* ptr to subtype supertype */
       int      gsn_num_sub;   /* number of subtype supertypes */
       struct   gen_sub_node      *gsn_next_genptr;
      };


struct ent_non_node
  /* structure def for each base-type nonentity node */
    {
     char     enn_name[ENLength + 1];
     char     enn_type;   /* either i(nteger), s(tring),
                  f(loat), e(numeration), or b(oolean) */
     int      enn_total_length; /* max length of base-type value */
     int      enn_range;  /* true or false depending on whether
                  there is a range.  If a range exists,
                  there must be two entries into ent_value */
     int      enn_num_values;    /* number of actual values */
     struct   ent_value    /*enn_value;  /* actual value of base-type */
     int      enn_constant;  /* boolean to refelect constant value */
     struct   ent_non_node *enn_next_node;
    };


struct sub_non_node
  /* structure def for each subtype nonentity node */
    {
     char     snn_name[ENLength + 1];
     char     snn_type; /* either i(nteger), s(tring),
                 f(loat), e(numeration), or b(oolean) */
     int      snn_total_length; /* max length of subtype value */
     int      snn_range;     /* true or false depending on whether
                   there is a range.  If a range exists,
                   there must be two entries into ent_value */
     int      snn_num_values; /* number of actual values */
     struct   ent_value       *snn_value; /* actual value of subtype */
     struct   sub_non_node    *snn_next_node;
    };
```

```
struct der_non_node
  /* structure def for each derived type nonentity node */
    {
    char      dnn_name[ENLength + 1];
    char      dnn_type;  /* either i(nteger), s(tring),
                    f(loat), e(numeration), or b(oolean) */
    int      dnn_total_length; /* max length of derived type value */
    int      dnn_range;   /* true or false depending on whether
                     there is a range. If a range exists,
                     there must be two entries into ent_value */
    int      dnn_num_values;   /* number of actual values */
    struct    ent_value    *dnn_value; /* actual value of derived type */
    struct    der_non_node *dnn_next_node;
    };


struct function_node
  /* structure definition for each function type declaration */
    {
     char     fn_name[ENLength+1];
     char     fn_type;         /* either f(loat), i(nteger), s(tring),
                         b(oolean), or e(numeration) */
     int      fn_range;        /* Boolean if range of values */
     int      fn_total_length; /* max length */
     int      fn_num_value;    /* number of actual values */
     struct   ent_value  *fn_value;  /* actual value */
     struct   ent_node   *fn_entptr; /* ptr to entity type */
     struct   gen_sub_node *fn_subptr; /* ptr to entity subtype */
     struct   ent_non_node *fn_nonentptr; /* ptr to nonentity type */
     struct   sub_non_node *fn_nonsubptr; /* ptr to nonentity subtype */
     struct   der_non_node *fn_nonderptr; /* ptr to nonentity dertype */
     int      fn_entnull; /* initialized false set true for no value */
     int      fn_unique;       /* init false - unique if true */
     struct   function_node *fn_next_fntptr;
    };


struct   user_info
         /* This structure is used to maintain information on all of the */
         /* current users of the particular interface.  The interface type */
         /* is determined by the li_info structure. */
         {
          char       ui_uid[UIDLength + 1]; /* The user id */
          union      li_info          ui_li_type; /* li is for language interface */
          struct     user_info        *ui_next_user;
         };
```

```
union   li_info
        /* This union is used to choose a particular data structure. */
        /* The data structure chosen is interface dependent, i.e.,   */
        /* li_sql is for the relational interface, li_dli is for the */
        /* hierarchical interface and li_dml is for the network int. */
      /* and li_dap is for the entity relationship interface.       */
        {
         struct    sql_info        li_sql;
         struct    dli_info        li_dli;
         struct    dml_info     li_dml;
       struct    dap_info    li_dap;
      };


struct  dap_info
     /*  The structure for info about the daplex request for a user */
      {
       struct     curr_db_info    dpi_curr_db; /* The current user */
       struct     file_info       dpi_file;  /* The dap files of request */
       struct     tran_info       dpi_dml_tran; /* The dml transactions */
       struct     ddl_info        *dpi_ddl_files; /* The abdl ddl files */
       int        dap_operation;  /* The operation to be performed */
       int        dap_answer;
       int        dap_error;
       int        dap_buff_count;
       union      kms_info       dpi_kms_data;
       union      kfs_info      dpi_kfs_data;
       union      kc_info       dpi_kc_data;
       };
```

APPENDIX B

THE STORAGE AND RETRIEVAL MODULES

A. STORAGE

```
                        /* this file is savefree.c */

#include<stdio.h>
#include"flags.def"
#include"licommdata.def"
#include"struct.def"
#include"dap.ext"

strfr_dap_db_list()
{
      /* begin strfr_dap_db_list */
        struct ent_dbid_node     *db_ptr;    /* ptr to the database list */
        struct ent_non_node      *non_ent_ptr; /* ptr to the nonentity node */
        struct ent_value *entval_ptr;  /* ptr to the entity value node */
        struct sub_non_node      *subnon_ptr; /* ptr to nonent subtype node */
      struct der_non_node     *dernon_ptr; /* ptr to derived subtype node */
        struct ent_node          *ent_node_ptr; /* ptr to the entity node */
        struct gen_sub_node      *gen_ptr;  /* ptr to the gen subtype node */
        struct overlap_ent_node *overlapent_ptr; /* ptr to the entity subtype */
        struct overlap_sub_node *overlapsub_ptr; /* ptr to the term subtype */
        struct function_node     *func_ptr;  /* ptr to the function node */
        FILE   *dap_fid;
        char temp_str[NUMDIGIT + 1];


                /* this function saves the entity/functional schema */
                /* back to a file and frees the list it occupied    */



#ifdef EnExFlag
        printf("Enter strfr_dap_db_list");
#endif

        if ((dap_fid = fopen (DAPDBSFname, "w")) == NULL)
              { /* begin if NULL */
                printf("Unable to open %s", DAPDBSFname);
                ring_the_bell();

#ifdef EnExFlag
        printf("Exit1 strfr_dap_db_list");
#endif
        return;
        } /* end if NULL */
```

```
db_ptr = dbs_dap_head_ptr.dn_dap;
while (db_ptr != NULL)
{ /* the database is stored to the file here */

    wr_ent_dbid_node(dap_fid,db_ptr);
    non_ent_ptr = db_ptr -> edn_nonentity;

  while (non_ent_ptr != NULL)
    { /* begin while non_ent_ptr != NULL */

        proc_ent_non_node(dap_fid, non_ent_ptr);
        non_ent_ptr = non_ent_ptr -> enn_next_node;

    } /* end while non_ent_ptr != NULL */

  ent_node_ptr = db_ptr -> edn_entity;
  num_to_str(db_ptr -> edn_num_ent, temp_str);
  writestr(dap_fid, temp_str);

  while (ent_node_ptr != NULL)
    { /* begin while ent_node_ptr != NULL) */

        wr_ent_node(dap_fid, ent_node_ptr);
        ent_node_ptr = ent_node_ptr -> en_next_ent;

    } /* end while ent_node_ptr != NULL */

  gen_ptr = db_ptr -> edn_subptr;
  num_to_str(db_ptr -> edn_num_gen, temp_str);
    writestr(dap_fid, temp_str);

  while (gen_ptr != NULL)
      { /* begin while gen_ptr != NULL */

        wr_gen_sub_node(dap_fid,gen_ptr);
        gen_ptr = gen_ptr -> gsn_next_genptr;

    } /* end while gen_ptr != NULL */

  gen_ptr = db_ptr -> edn_subptr;

  while(gen_ptr != NULL)
    { /* begin while gen_ptr != NULL */

      num_to_str(gen_ptr -> gsn_num_ent, temp_str);
        writestr (dap_fid, temp_str);
        overlapent_ptr = gen_ptr -> gsn_entptr;

      while( overlapent_ptr != NULL)
            { /* begin while overlapent_ptr != NULL */

          writestr(dap_fid, overlapent_ptr -> oen_name -> en_name);
          overlapent_ptr = overlapent_ptr -> oen_next_name;

        } /* end while overlapent_ptr != NULL */
```

```
      num_to_str(gen_ptr -> gsn_num_sub, temp_str);
        writestr(dap_fid, temp_str);
        overlapsub_ptr = gen_ptr -> gsn_subptr;

        while (overlapsub_ptr != NULL)
            { /* begin while overlapsub_ptr != NULL */

            writestr(dap_fid, overlapsub_ptr -> osn_name -> gsn_name);
            overlapsub_ptr = overlapsub_ptr ->osn_next_name;

          } /* end while overlapsub_ptr != NULL */

   gen_ptr = gen_ptr -> gsn_next_genptr;

  } /* end while gen_ptr != NULL */

/* Process the sub non nodes */

 subnon_ptr = db_ptr -> edn_nonsubptr;
 num_to_str (db_ptr -> edn_num_nonsub, temp_str);
 writestr (dap_fid, temp_str);

 while (subnon_ptr != NULL)
   { /* begin while subnon_ptr <> NULL */

          proc_sub_non_node (dap_fid,subnon_ptr);
     subnon_ptr = subnon_ptr -> snn_next_node;

   } /* end while subnon_ptr <> NULL */


/* Process the derived nodes */

 dernon_ptr = db_ptr -> edn_nonderptr;
 num_to_str(db_ptr -> edn_num_der, temp_str);
 writestr(dap_fid, temp_str);

 while (dernon_ptr != NULL)
   { /* begin while dernon_ptr <> NULL */

     proc_der_non_node (dap_fid,dernon_ptr);
     dernon_ptr = dernon_ptr -> dnn_next_node;

    } /* end while dernon_ptr <> NULL */

/* Process the Ent function nodes */

   ent_node_ptr = db_ptr -> edn_entity;

 while (ent_node_ptr != NULL)
   { /* begin while ent_node_ptr != NULL */

        wr_all_ent_node(dap_fid, ent_node_ptr);
        ent_node_ptr = ent_node_ptr -> en_next_ent;

   } /* end while ent_node_ptr != NULL */
```

```
        gen_ptr = db_ptr -> edn_subptr;

       while (gen_ptr != NULL)
         { /* begin while gen_ptr != NULL */

           func_ptr = gen_ptr -> gsn_ftnptr;

             while (func_ptr != NULL)
                   { /* begin while func_ptr != NULL */

                   proc_function_node(dap_fid,func_ptr);
                   gen_ptr->gsn_ftnptr = func_ptr->fn_next_fntptr;
                 free_function_node(func_ptr);
                 func_ptr = gen_ptr -> gsn_ftnptr;

               } /* end while func_ptr != NULL */

           gen_ptr = gen_ptr -> gsn_next_genptr;

         } /* end while gen_ptr != NULL */

        db_ptr = db_ptr -> edn_next_db;

      } /* end while db_ptr != NULL */

    putc('$', dap_fid);
    putc('', dap_fid);


#ifdef EnExFlag
    printf("Exit strfr_dap_db_list");
#endif

} /* end strfr_dap_db_list */

wr_ent_dbid_node( ffid, db_ptr)
    FILE    *ffid;
      struct    ent_dbid_node    *db_ptr;
{
      char temp_str[NUMDIGIT + 1];
      /* this function writes the database */
    /* structure's contents to the save file */

#ifdef EnExFlag
      printf("Enter wr_ent_dbid_node");
#endif

      writestr(ffid, db_ptr -> edn_name);
      num_to_str(db_ptr -> edn_num_nonent, temp_str);
      writestr(ffid, temp_str);

#ifdef EnExFlag
      printf("Exit wr_ent_dbid_node");
#endif
} /* end wr_ent_dbid_node */
```

```
proc_ent_non_node(fid, non_ptr)
    FILE     *fid;
      struct   ent_non_node    *non_ptr;
      { /* begin proc_ent_non_node */
            struct      ent_value       *val_ptr;
              char      temp_str[NUMDIGIT + 1];

#ifdef EnExFlag
      printf("Enter proc_ent_non_node");
#endif

      writestr(fid, non_ptr -> enn_name);
      putc(non_ptr -> enn_type, fid);
      putc('', fid);
      num_to_str(non_ptr -> enn_total_length, temp_str);
      writestr(fid, temp_str);
      num_to_str(non_ptr -> enn_range, temp_str);
      writestr(fid, temp_str);
      num_to_str(non_ptr -> enn_constant, temp_str);
      writestr(fid, temp_str);
      num_to_str(non_ptr -> enn_num_values, temp_str);
      writestr(fid, temp_str);
      val_ptr = non_ptr -> enn_value;

      while (val_ptr != NULL)
        { /* begin while val_ptr <> NULL */

        writestr(fid,val_ptr -> ev_value);
        val_ptr = val_ptr -> ev_next_value;

      } /* end while val_ptr <> NULL */

#ifdef EnExFlag
      printf("Exit proc_ent_non_node");
#endif

      } /* end proc_ent_non_node */

wr_ent_node(fid,ent_ptr)
    FILE     *fid;
      struct   ent_node         *ent_ptr;
      { /* begin wr_ent_node */
            char        temp_str[NUMDIGIT + 1];

#ifdef EnExFlag
      printf("Enter wr_ent_node");
#endif

      writestr(fid, ent_ptr -> en_name);
      num_to_str(ent_ptr -> en_num_funct, temp_str);
      writestr(fid, temp_str);
    num_to_str(ent_ptr -> en_terminal, temp_str);
    writestr(fid, temp_str);


#ifdef EnExFlag
```

```
        printf("Exit wr_ent_node");
#endif
        } /* end wr_ent_node */


wr_gen_sub_node(fid,gs_ptr)
      FILE        *fid;
        struct  gen_sub_node     *gs_ptr;
        { /* begin wr_gen_sub_node */
        char      temp_str[NUMDIGIT + 1];

#ifdef EnExFlag
        printf("Enter wr_gen_sub_node");
#endif

        writestr(fid,gs_ptr -> gsn_name);
        num_to_str(gs_ptr -> gsn_num_funct, temp_str);
        writestr(fid, temp_str);
      num_to_str(gs_ptr -> gsn_terminal, temp_str);
      writestr(fid, temp_str);


#ifdef EnExFlag
        printf("Exit wr_gen_sub_node");
#endif
        } /* end wr_gen_sub_node */

proc_sub_non_node(fid, sub_ptr)
      FILE        *fid;
        struct   sub_non_node    *sub_ptr;
        { /* begin proc_sub_non_node */
            FILE          *fid;
        struct   ent_value          *val_ptr;
                char      temp_str[NUMDIGIT + 1];

#ifdef EnExFlag
        printf("Enter proc_sub_non_node");
#endif

        writestr(fid, sub_ptr -> snn_name);
        putc (sub_ptr -> snn_type, fid);
        putc('', fid);
        num_to_str(sub_ptr -> snn_total_length, temp_str);
        writestr(fid,temp_str);
      num_to_str(sub_ptr -> snn_range, temp_str);
      writestr(fid, temp_str);
        val_ptr = sub_ptr -> snn_value;

        while (val_ptr != NULL)
          { /* begin while val_ptr <> NULL */

          writestr(fid, val_ptr -> ev_value );
            val_ptr = val_ptr -> ev_next_value;

       } /* end while val_ptr <> NULL */
```

```
#ifdef EnExFlag
        printf("Exit proc_sub_non_node");
#endif
        } /* end proc_sub_non_node */

proc_der_non_node(fid,der_ptr)
        FILE    *fid;
      struct  der_non_node        *der_ptr;
      { /* begin proc_der_non_node */
                struct   ent_value         *val_ptr;
            char temp_str[NUMDIGIT + 1];

#ifdef EnExFlag
        printf("Enter proc_der_non_node");
#endif

        writestr(fid, der_ptr -> dnn_name);
      putc (der_ptr -> dnn_type, fid);
        putc('', fid);
      num_to_str(der_ptr -> dnn_total_length, temp_str);
      writestr(fid, temp_str);
      num_to_str(der_ptr -> dnn_range, temp_str);
      writestr(fid, temp_str);
      val_ptr = der_ptr -> dnn_value;

      while (val_ptr != NULL)
        { /* begin while val_ptr <> NULL */

          writestr(fid, val_ptr -> ev_value);
          val_ptr = val_ptr -> ev_next_value;

        } /* end while val_ptr <> NULL */

#ifdef EnExFlag
        printf("Exit proc_der_non_node");
#endif

        } /* end proc_der_non_node */

wr_all_ent_node(fid,ent_ptr)
        FILE    *fid;
        struct   ent_node          *ent_ptr;
        { /* begin wr_all_ent_node */
                struct   function_node  *funct_ptr;
                char     temp_str[NUMDIGIT + 1];

#ifdef EnExFlag
        printf("Enter wr_all_ent_node");
#endif

        funct_ptr = ent_ptr -> en_ftnptr;

        while (funct_ptr != NULL)
          { /* begin while funct_ptr <> NULL */

          proc_function_node(fid,funct_ptr);
```

```
      ent_ptr -> en_ftnptr = funct_ptr -> fn_next_fntptr;
      free_function_node(funct_ptr);
      funct_ptr = ent_ptr -> en_ftnptr;

    } /* end while funct_ptr <> NULL */

#ifdef EnExFlag
        printf("Exit wr_all_ent_node");
#endif
        } /* end wr_all_ent_node */


proc_function_node(fid,fptr)
        FILE    *fid;
        struct  function_node   *fptr;
        { /* begin proc_function_node */
                struct  ent_value       *val_ptr;
                struct  ent_node*eptr;
                struct  gen_sub_node    *gsptr;
                struct  ent_non_node    *enptr;
                struct  sub_non_node    *snptr;
            struct  der_non_node    *dnptr;
                char    temp_str[NUMDIGIT + 1];

#ifdef EnExFlag
        printf("Enter proc_function_node");
#endif

        writestr(fid, fptr -> fn_name);
        putc(fptr -> fn_type, fid);
        putc('', fid);
        num_to_str(fptr -> fn_range, temp_str);
        writestr(fid, temp_str);
        num_to_str(fptr -> fn_total_length, temp_str);
        writestr(fid, temp_str);
        num_to_str(fptr -> fn_num_value, temp_str);
        writestr(fid, temp_str);
        val_ptr = fptr -> fn_value;

        while (val_ptr != NULL)
          { /* begin while val_ptr <> NULL */

          writestr ( fid, val_ptr -> ev_value);
          val_ptr = val_ptr -> ev_next_value;

        } /* end while val_ptr <> NULL */

        eptr = fptr -> fn_entptr;

      if(eptr != NULL)
        writestr(fid, eptr -> en_name);
      else
        writestr(fid, "^");

        gsptr = fptr -> fn_subptr;
```

```
        if(gsptr != NULL)
            writestr(fid, gsptr -> gsn_name);
        else
          writestr(fid, "^");

        enptr = fptr -> fn_nonentptr;

        if(enptr != NULL)
          writestr(fid, enptr -> enn_name);
        else
          writestr(fid, "^");

        snptr = fptr -> fn_nonsubptr;

        if(snptr != NULL)
          writestr(fid, snptr -> snn_name);
        else
          writestr(fid, "^");

        dnptr = fptr -> fn_nonderptr;

        if(dnptr != NULL)
          writestr(fid, dnptr -> dnn_name);
        else
          writestr(fid, "^");

            num_to_str(fptr -> fn_entnull, temp_str);
            writestr(fid, temp_str);
              num_to_str(fptr -> fn_unique, temp_str);
              writestr(fid, temp_str);


#ifdef EnExFlag
          printf("Exit proc_function_node");
#endif
          } /* end proc_function_node */


free_function_node(func_ptr)
          struct   function_node   *func_ptr;
          { /* begin free_function_node */

#ifdef EnExFlag
          printf("Enter free_function_node");
#endif

          strcpy(func_ptr -> fn_name, " ");
          func_ptr -> fn_type = ' ';
          func_ptr -> fn_range = 0;
          func_ptr -> fn_total_length = 0;
          func_ptr -> fn_num_value = 0;
          func_ptr -> fn_value = NULL;
          func_ptr -> fn_entptr = NULL;
          func_ptr -> fn_subptr = NULL;
          func_ptr -> fn_nonentptr = NULL;
          func_ptr -> fn_nonsubptr = NULL;
```

```
        func_ptr -> fn_nonderptr = NULL;
          func_ptr -> fn_entnull = 0;
          func_ptr -> fn_unique = 0;
          func_ptr -> fn_next_fntptr = NULL;

#ifdef EnExFlag
          printf("Exit free_function_node");
#endif
          } /* end free_function_node */
```

B. RETRIEVAL

```
                              /* This file is makelist.c */

#include <stdio.h>
#include "flags.def"
#include "licommdata.def"
#include "struct.def"
#include "dap.ext"

creat_dap_db_list()
{
struct  ent_dbid_node *new_db_ptr, *db_ptr; /* ptr to the enty-relationp node */
struct  ent_non_node  *new_non_ent_ptr, *non_ent_ptr; /* ptr to nonenty node */
struct  ent_value *new_entval_ptr, *entval_ptr; /* ptr to the enty value node */
struct  sub_non_node *new_subnon_ptr, *subnon_ptr; /*ptr to sub_nonenty node */
struct  ent_node  *new_ent_node_ptr, *ent_node_ptr; /* ptr to the entity node */
struct  gen_sub_node *new_gen_ptr, *gen_ptr; /* ptr to gen super,subtype node */
struct  overlap_ent_node *new_overlapent_ptr, *overlapent_ptr;
struct  overlap_sub_node *new_overlapsub_ptr, *overlapsub_ptr;
struct  function_node *new_func_ptr, *func_ptr; /* ptr to the function node */
struct  der_non_node  *new_dernon_ptr, *dernon_ptr; /* ptr to nonent der node */
int  ed_count, ent_count, funct_count, num_val;  /* counters */
int  gen_sub_count, super_count, sub_super_count;  /* counters */
int  non_sub_count, non_der_count, enum_count;    /* counters */
int  done_flag, first_db, first_nonnode;          /* booleans */
int  first_enum, first_node, first_func;          /* booleans */
int  first_value, first_gen_sub, first_super;     /* booleans */
int  first_sub_super, first_non_sub;              /* booleans */
int  first_non_der;                               /* boolean  */
struct  ent_dbid_node  *rd_ent_dbid_node();
struct  ent_non_node   *rd_ent_non_node();
struct  ent_value      *rd_ent_value();
struct  sub_non_node   *rd_sub_non_node();
struct  ent_node       *rd_ent_node();
struct  gen_sub_node   *rd_gen_sub_node();
struct  overlap_ent_node *rd_overlap_ent_node();
struct  overlap_sub_node *rd_overlap_sub_node();
struct  function_node  *rd_function_node();
struct  der_non_node   *rd_der_non_node();
 FILE    *dap_fid;
 char    temp_str[NUMDIGIT + 1];
```

```
/* This function retrieves and recreates the schema from the stored file */

#iflef EnExFlag
      printf ("Enter create_dap_db_list");
#endif

  if ( (dap_fid = fopen( DAPDBSFname, "r") ) == NULL)
    {
      printf ("Unable to open file %s", DAPDBSFname);
      ring_the_bell();

#iflef EnExFlag
      printf ("Exit1 creat_dap_db_list");
#endif
      return;
    }
  done_flag = FALSE;
  first_db = TRUE;
  while ( done_flag != TRUE )
    {
      /* the schema nodes are allocated and filled here */
      new_db_ptr = rd_ent_dbid_node( dap_fid, &done_flag);
      if ( done_flag != TRUE )
        {
          if ( first_db == TRUE )
            {
              /* special case of accessing the first entity relationship */
              dbs_dap_head_ptr.dn_dap = new_db_ptr;
              db_ptr = new_db_ptr;
              first_db = TRUE;
            }
          else
            {
              db_ptr->edn_next_db = new_db_ptr;
              db_ptr = new_db_ptr;
            }

          first_nonnode = TRUE;
          ed_count = db_ptr->edn_num_nonent;
          while ( ed_count != 0 )
            {
              /* the nonentity nodes are allocated and filled here */
              new_non_ent_ptr = rd_ent_non_node(dap_fid);
              if ( first_nonnode == TRUE )
                {
                  /* special case for first nonentity */
                  db_ptr->edn_nonentity = new_non_ent_ptr;
                  non_ent_ptr = new_non_ent_ptr;
                  first_nonnode = FALSE;
                }
```

```
  else
    {
      non_ent_ptr->enn_next_node = new_non_ent_ptr;
      non_ent_ptr = new_non_ent_ptr;
    }
  first_enum = TRUE;
  enum_count = non_ent_ptr->enn_num_values;
  while ( enum_count != 0 )
    {
      /* the actual value nodes are allocated and filled here */
      new_entval_ptr = rd_ent_value(dap_fid,
                                    non_ent_ptr->enn_total_length);
      if ( first_enum == TRUE )
        {
          /* special case of first actual value */
          non_ent_ptr->enn_value = new_entval_ptr;
          entval_ptr = new_entval_ptr;
          first_enum = FALSE;
        }
      else
        {
          entval_ptr->ev_next_value = new_entval_ptr;
          entval_ptr = new_entval_ptr;
        }
      --enum_count;
    } /* end value loop */

  --ed_count;
 } /* end base type nonentity loop */

first_node = TRUE;
readstr(dap_fid,temp_str);
db_ptr->edn_num_ent = str_to_num(temp_str);
ent_count = db_ptr->edn_num_ent;
while ( ent_count != 0 )
 {
  /* the entity nodes are allocated and filled in here */
  new_ent_node_ptr = rd_ent_node(dap_fid);
  if (first_node == TRUE )
    {
      /* special case of first entity node */
      db_ptr->edn_entity = new_ent_node_ptr;
      ent_node_ptr = new_ent_node_ptr;
      first_node = FALSE;
    }
  else
    {
      ent_node_ptr->en_next_ent = new_ent_node_ptr;
      ent_node_ptr = new_ent_node_ptr;
    }
  ent_count--;
 }

first_gen_sub = TRUE;
readstr(dap_fid,temp_str);
db_ptr->edn_num_gen = str_to_num(temp_str);
```

```
    gen_sub_count = db_ptr->edn_num_gen;
    while ( gen_sub_count != 0 )
      {
        /* the gen subtype nodes are allocated and filled here */
        new_gen_ptr = rd_gen_sub_node(dap_fid);
        if ( first_gen_sub == TRUE )
          {
            /* special case of first generalization node */
            db_ptr->edn_subptr = new_gen_ptr;
            gen_ptr = new_gen_ptr;
            first_gen_sub = FALSE;
          }
        else
          {
            gen_ptr->gsn_next_genptr = new_gen_ptr;
            gen_ptr = new_gen_ptr;
          }
        gen_sub_count--;
      }

    /* Process the overlap nodes */

  gen_ptr = db_ptr -> edn_subptr;
  while (gen_ptr != NULL)
    { /* begin while gen_ptr <> NULL */
    first_super = TRUE;
    readstr(dap_fid,temp_str);
    gen_ptr->gsn_num_ent = str_to_num(temp_str);
    super_count = gen_ptr->gsn_num_ent;
    while ( super_count != 0 )
      {
        /* the subtypes with one or more entity supertypes */
        /* nodes are allocated and filled here             */
        new_overlapent_ptr = rd_overlap_ent_node(dap_fid,
                                        db_ptr->edn_entity);
        if ( first_super == TRUE )
          {
            /* the special case of the first overlap ent node */
            gen_ptr->gsn_entptr = new_overlapent_ptr;
            overlapent_ptr = new_overlapent_ptr;
            first_super = FALSE;
          }
        else
          {
            overlapent_ptr->oen_next_name = new_overlapent_ptr;
            overlapent_ptr = new_overlapent_ptr;
          }
        --super_count;
      } /* end super type node */

    first_sub_super = TRUE;
    readstr(dap_fid,temp_str);
    gen_ptr->gsn_num_sub   = str_to_num(temp_str);
    sub_super_count = gen_ptr->gsn_num_sub;
    while ( sub_super_count != 0 )
      {
```

```
      /* the subtype supertypes are allocated here */
      new_overlapsub_ptr = rd_overlap_sub_node(dap_fid,
                                        db_ptr->edn_subptr);
      if ( first_sub_super == TRUE )
        {
          /* special case of first overlapsub node */
          gen_ptr->gsn_subptr = new_overlapsub_ptr;
          overlapsub_ptr = new_overlapsub_ptr;
          first_sub_super = FALSE;
        }
      else
        {
          overlapsub_ptr->osn_next_name = new_overlapsub_ptr;
          overlapsub_ptr = new_overlapsub_ptr;
        }
      --sub_super_count;
    } /* end overlapsub loop */

  gen_ptr = gen_ptr -> gsn_next_genptr;

 }
   /* Process the sub non nodes */

 readstr(dap_fid,temp_str);
 db_ptr->edn_num_nonsub = str_to_num(temp_str);
 first_non_sub = TRUE;
 non_sub_count = db_ptr->edn_num_nonsub;
 while ( non_sub_count != 0 )
   {
     /* the nonentity subtype nodes are allocated and filled */
     new_subnon_ptr = rd_sub_non_node(dap_fid);
     if ( first_non_sub == TRUE )
       {
         /* special case of first nonentity subtype node */
         db_ptr->edn_nonsubptr = new_subnon_ptr;
         subnon_ptr = new_subnon_ptr;
         first_non_sub = FALSE;
       }
     else
       {
         subnon_ptr->snn_next_node = new_subnon_ptr;
         subnon_ptr = new_subnon_ptr;
       }
     first_value = TRUE;
     num_val = subnon_ptr->snn_num_values;
     while ( num_val != 0 )
       {
         /* the value nodes are allocated and filled here */
         new_entval_ptr = rd_ent_value(dap_fid,
                                  subnon_ptr->snn_total_length);
         if ( first_value == TRUE )
           {
             /* special case of first actual value */
             subnon_ptr->snn_value = new_entval_ptr;
             entval_ptr = new_entval_ptr;
```

```
          first_value = FALSE;
        }
      else
        {
          entval_ptr->ev_next_value = new_entval_ptr;
          entval_ptr = new_entval_ptr;
        }
      --num_val;
    } /* end actual value loop */

  --non_sub_count;
} /* end subtype nonentity loop */


  /* Process the derived nodes */

readstr(dap_fid,temp_str);
db_ptr->edn_num_der = str_to_num(temp_str);
first_non_der = TRUE;
non_der_count = db_ptr->edn_num_der;
while ( non_der_count != 0 )
  {
    /* the nonentity derived types are allocated and filled here */
    new_dernon_ptr = rd_der_non_node(dap_fid);
    if ( first_non_der == TRUE )
      {
        /* special case of first derived type nonentity node */
        db_ptr->edn_nonderptr = new_dernon_ptr;
        dernon_ptr = new_dernon_ptr;
        first_non_der = FALSE;
      }
    else
      {
        dernon_ptr->dnn_next_node = new_dernon_ptr;
        dernon_ptr = new_dernon_ptr;
      }

    first_value = TRUE;
    num_val = dernon_ptr->dnn_num_values;
    while ( num_val != 0 )
      {
        /* the value nodes are allocated and filled here */
        new_entval_ptr = rd_ent_value(dap_fid,
                               dernon_ptr->dnn_total_length);
        if ( first_value == TRUE )
          {
            /* special case of first actual value */
            dernon_ptr->dnn_value = new_entval_ptr;
            entval_ptr = new_entval_ptr;
            first_value = FALSE;
          }
        else
          {
            entval_ptr->ev_next_value = new_entval_ptr;
            entval_ptr = new_entval_ptr;
          }
```

```
        --num_val;
      } /* end actual value loop */

    --non_der_count;
  } /* end derived type non entity loop */

   /* NOW PROCESS THE FUNCTION NODES */

/* First, for entity nodes */

ent_node_ptr = db_ptr->edn_entity;
while ( ent_node_ptr != NULL )
  {
    first_func = TRUE;
    funct_count = ent_node_ptr->en_num_funct;
    while ( funct_count != 0 )
      {
        /* function type nodes are allocated and filled here */
        new_func_ptr = rd_function_node(dap_fid,db_ptr);
        if ( first_func == TRUE )
          {
            /* the special case of first function node */
            ent_node_ptr->en_ftnptr = new_func_ptr;
            func_ptr = new_func_ptr;
            first_func = FALSE;
          }
        else
          {
            func_ptr->fn_next_fntptr = new_func_ptr;
            func_ptr = new_func_ptr;
          }
        --funct_count;
      } /* end function loop */
    ent_node_ptr = ent_node_ptr->en_next_ent;
  } /* end while loop for ent function nodes */

/* Now Process the gen sub node function nodes */

gen_ptr = db_ptr->edn_subptr;
while ( gen_ptr != NULL )
  {
      first_func = TRUE;
      funct_count = gen_ptr->gsn_num_funct;
      while ( funct_count != 0 )
        {
          /* the function type nodes are allocated and filled here */
          new_func_ptr = rd_function_node(dap_fid,db_ptr);
          if ( first_func == TRUE )
            {
              /* the special case of the first function node */
              gen_ptr->gsn_ftnptr = new_func_ptr;
              func_ptr = new_func_ptr;
              first_func = FALSE;
            }
          else
            {
```

```
                func_ptr->fn_next_fntptr = new_func_ptr;
                func_ptr = new_func_ptr;
              }
            funct_count--;
          } /* end function loop */
      gen_ptr = gen_ptr->gsn_next_genptr;
     } /* end while loop for gen sub function nodes */

    } /* end if done_flag != TRUE loop */

  } /* end shema makelist loop */
#ifdef EnExFlag
    printf ("Exit2 creat_dap_db_list");
#endif
} /* End creat_dap_db_list */

static  struct  ent_dbid_node   *rd_ent_dbid_node ( fid, flag )
      FILE    *fid;
      int     *flag;

 {
   struct  ent_dbid_node  *db_ptr,    /* pointer to database node */
                   * ent_dbid_node_alloc( ); /* pointer to newly */
                             /* allocated database node */
   char  temp_str [NUMDIGIT + 1];      /* temp string to hold file ID */

#ifdef EnExFlag
      printf("Enter rd_ent_dbid_node");
#endif

    /* this function allocates a new database node and returns a pointer */
    /* to it */

    /* a new database node is established and ptrs are initialized */

    db_ptr = ent_dbid_node_alloc( );
    db_ptr->edn_nonentity = NULL;
    db_ptr->edn_entity = NULL;
    db_ptr->edn_subptr = NULL;
    db_ptr->edn_next_db = NULL;
    db_ptr->edn_nonsubptr = NULL;
    db_ptr->edn_nonderptr = NULL;

    readstr(fid,db_ptr->edn_name);
    if ( db_ptr->edn_name[0] == '$' )
      {
        /* when file becomes empty */
        *flag = TRUE;
        free (db_ptr);

#ifdef EnExFlag
      printf("Exit1 ent_dbid_node");
#endif
      return(NULL);
      }
    else
```

```
        {
          readstr(fid,temp_str);
          db_ptr->edn_num_nonent = str_to_num(temp_str);

#ifdef EnExFlag
        printf("Exit2 rd_ent_dbid_node");
#endif
          return(db_ptr);
        }
   } /* end rd_ent_dbid_node */


 static struct  ent_non_node  *rd_ent_non_node(fid)

    FILE  *fid;
  {
    struct ent_non_node    *non_ent_ptr,  /* pointer to base type */
                                          /* nonentity node       */
                  *ent_non_node_alloc(); /* pointer to newly allocated */
                                          /* nonentity node            */

    char  temp_str[NUMDIGIT + 1];        /* temp string to read fields   */


       /* this function allocates a new base type nonentity node and */
       /* returns a pointer to it                                     */

#ifdef EnExFlag
        printf ("Enter rd_ent_non_node");
#endif

       /* get new base type nonentity node and initialize pointers */

       non_ent_ptr = ent_non_node_alloc( );
       non_ent_ptr->enn_value = NULL;
       non_ent_ptr->enn_next_node = NULL;

       /* now the node is filled in by reading the file */

       readstr(fid,non_ent_ptr->enn_name);
       readstr(fid,temp_str);
       non_ent_ptr->enn_type = temp_str[0];
       readstr(fid,temp_str);
       non_ent_ptr->enn_total_length = str_to_num(temp_str);
       readstr(fid,temp_str);
       non_ent_ptr->enn_range = str_to_num(temp_str);
       readstr(fid,temp_str);
       non_ent_ptr->enn_constant = str_to_num(temp_str);
       readstr(fid,temp_str);
       non_ent_ptr->enn_num_values = str_to_num(temp_str);

#ifdef EnExFlag
        printf("Exit rd_ent_non_node");
#endif
       return(non_ent_ptr);
   } /* end rd_ent_non_node */
```

```
static struct  ent_value    *rd_ent_value(fid, length)
  FILE  *fid;
  int   length;

  {
    struct  ent_value   *entval_ptr,   /* pointer to def for value */
                *ent_value_alloc( ); /* pointer fo newly allocated */
                                /* value node               */
    char  temp_str[NUMDIGIT + 1];   /* temp string to read fields    */
    char  *var_str_alloc();

    /* this function allocates a new value node and returns a pointer */
    /* to it                                                         */

#ifdef EnExFlag
      printf("Enter rd_ent_value_node");
#endif

    /* get the new value node and initialize ptrs */

      entval_ptr = ent_value_alloc( );
      entval_ptr->ev_next_value = NULL;

    /* now value node is filled in by reading the file */

      entval_ptr->ev_value = var_str_alloc(length + 1);
      readstr(fid,entval_ptr->ev_value);

#ifdef EnExFlag
      printf("Exit rd_ent_value_node");
#endif

      return(entval_ptr);
  } /* end rd_ent_value */


static struct ent_node   *rd_ent_node(fid)
   FILE  *fid;
  {
    struct ent_node   *ent_node_ptr,   /* pointer to entity node */
             *ent_node_alloc( );    /* pointer to newly allocated */
                               /* entity node               */
    char temp_str[NUMDIGIT + 1];

    /* this function allocates a new entity node and returns a pointer */
    /* to it                                                          */

#ifdef EnExFlag
      printf("Enter rd_ent_node");
#endif

      /* get new entity node and initialize values */
```

```
        ent_node_ptr = ent_node_alloc( );
        ent_node_ptr->en_ftnptr = NULL;
        ent_node_ptr->en_next_ent = NULL;

        /* now the entity node is filled in by reading the file */

        readstr(fid,ent_node_ptr->en_name);
        readstr(fid,temp_str);
        ent_node_ptr->en_num_funct = str_to_num(temp_str);
        readstr(fid,temp_str);
        ent_node_ptr->en_terminal = str_to_num(temp_str);

#ifdef EnExFlag
        printf("Exit rd_ent_node");
#endif

        return(ent_node_ptr);
    } /* end rd_ent_node */


  static struct  function_node  *rd_function_node(fid, db_ptr)
    FILE  *fid;
    struct  ent_dbid_node   *db_ptr;
    {
     struct function_node  *func_ptr,  /* pointer to function type node */
                *function_node_alloc( ); /* pointer to newly allocated */
                                          /* function type node        */
     char temp_str[NUMDIGIT + 1];
     char name_str[ENLength + 1];
     int  num_val,
          first_value;
     struct  ent_value   *entval_ptr,
                         *new_entval_ptr,
                         *rd_ent_value();
     struct  ent_node    *ent_ptr;
     struct  gen_sub_node *sub_ptr;
     struct  ent_non_node *enon_ptr;
     struct  sub_non_node *non_ptr;
     struct  der_non_node *der_ptr;
     int     done_flag;

    /* this function allocates a new function node and returns a pointer */
    /* to it                                                             */

#ifdef EnExFlag
        printf("Enter rd_function_node");
#endif

        /* get new function node and initialize values */

        func_ptr = function_node_alloc( );
        func_ptr->fn_value = NULL;
        func_ptr->fn_entptr = NULL;
        func_ptr->fn_subptr = NULL;
        func_ptr->fn_nonentptr = NULL;
        func_ptr->fn_nonsubptr = NULL;
```

```
func_ptr->fn_nonderptr = NULL;
func_ptr->fn_next_fntptr = NULL;

/* now the function node is filled in by reading the file */

readstr(fid,func_ptr->fn_name);
readstr(fid,temp_str);
func_ptr->fn_type = temp_str[0];
readstr(fid,temp_str);
func_ptr->fn_range = str_to_num(temp_str);
readstr(fid,temp_str);
func_ptr->fn_total_length = str_to_num(temp_str);
readstr(fid,temp_str);
func_ptr->fn_num_value = str_to_num(temp_str);

first_value = TRUE;
num_val = func_ptr->fn_num_value;
while ( num_val != 0 )
  {
    /* value nodes are allocated and filled here */
    new_entval_ptr = rd_ent_value(fid,
                                  func_ptr->fn_total_length);
    if ( first_value == TRUE )
      {
        /* special case of first value */
        func_ptr->fn_value = new_entval_ptr;
        entval_ptr = new_entval_ptr;
        first_value = FALSE;
      }
    else
      {
        entval_ptr->ev_next_value = new_entval_ptr;
        entval_ptr = new_entval_ptr;
      }
    --num_val;
  } /* end value loop */

readstr(fid, name_str);
if (name_str[0] != '^')
  {
    done_flag = FALSE;
    ent_ptr = db_ptr->edn_entity;
    while (done_flag == FALSE)
      if (strcmp(name_str, ent_ptr->en_name) == 0)
       {
        done_flag = TRUE;
        func_ptr->fn_entptr = ent_ptr;
       }
      else
       {
        ent_ptr = ent_ptr->en_next_ent;
        if (ent_ptr == NULL) done_flag = TRUE;
       }
  }

readstr(fid, name_str);
```

```
if (name_str[0] != '^')
  {
    done_flag = FALSE;
    sub_ptr = db_ptr->edn_subptr;
    while (done_flag == FALSE)
      if (strcmp(name_str, sub_ptr->gsn_name) == 0)
       {
        done_flag = TRUE;
        func_ptr->fn_subptr = sub_ptr;
       }
      else
       {
        sub_ptr = sub_ptr->gsn_next_genptr;
        if (sub_ptr == NULL) done_flag = TRUE;
       }
  }

readstr(fid, name_str);
if (name_str[0] != '^')
  {
    done_flag = FALSE;
    enon_ptr = db_ptr->edn_nonentity;
    while (done_flag == FALSE)
      if (strcmp(name_str, enon_ptr->enn_name) == 0)
       {
        done_flag = TRUE;
        func_ptr->fn_nonentptr = enon_ptr;
       }
      else
       {
        enon_ptr = enon_ptr->enn_next_node;
        if (enon_ptr == NULL) done_flag = TRUE;
       }
  }

readstr(fid, name_str);
if (name_str[0] != '^')
  {
    done_flag = FALSE;
    non_ptr = db_ptr->edn_nonsubptr;
    while (done_flag == FALSE)
      if (strcmp(name_str, non_ptr->snn_name) == 0)
       {
        done_flag = TRUE;
        func_ptr->fn_nonsubptr = non_ptr;
       }
      else
       {
        non_ptr = non_ptr->snn_next_node;
        if (non_ptr == NULL) done_flag = TRUE;
       }
  }

readstr(fid, name_str);
if (name_str[0] != '^')
  {
```

```
          done_flag = FALSE;
          der_ptr = db_ptr->edn_nonderptr;
          while (done_flag == FALSE)
            if (strcmp(name_str, der_ptr->dnn_name) == 0)
             {
              done_flag = TRUE;
              func_ptr->fn_nonderptr = der_ptr;
             }
            else
             {
              der_ptr = der_ptr->dnn_next_node;
              if (der_ptr == NULL) done_flag = TRUE;
             }
        }

      readstr(fid,temp_str);
      func_ptr->fn_entnull = str_to_num(temp_str);
      readstr(fid,temp_str);
      func_ptr->fn_unique = str_to_num(temp_str);

#ifdef EnExFlag
      printf("Exit rd_function node");
#endif
      return(func_ptr);
  }  /* end rd_function_node */


 static struct gen_sub_node  *rd_gen_sub_node(fid)
   FILE  *fid;
  {
   struct gen_sub_node   *gen_ptr,   /* pointer to generalization node */
           *gen_sub_node_alloc( ); /* pointer to newly allocated     */
                                   /* generalization node            */
   char temp_str[NUMDIGIT + 1];

   /* this function allocates a new generalization node and returns a  */
   /* pointer to it                                                   */

#ifdef EnExFlag
      printf("Enter rd_gen_sub_node");
#endif

     /* get new generalization node and initialize ptrs */

     gen_ptr = gen_sub_node_alloc( );
     gen_ptr->gsn_entptr = NULL;
     gen_ptr->gsn_ftnptr = NULL;
     gen_ptr->gsn_subptr = NULL;
     gen_ptr->gsn_next_genptr = NULL;

     /* now the generalization node is filled in by reading the file */

     readstr(fid,gen_ptr->gsn_name);
     readstr(fid,temp_str);
     gen_ptr->gsn_num_funct = str_to_num(temp_str);
     readstr(fid,temp_str);
```

```
      gen_ptr->gsn_terminal = str_to_num(temp_str);

#ifdef EnExFlag
      printf("Exit rd_gen_sub_node");
#endif

      return(gen_ptr);
   } /* end rd_gen_sub_node */


  static struct overlap_ent_node  *rd_overlap_ent_node(fid, ent_ptr)
    FILE  *fid;
    struct  ent_node *ent_ptr;
   {
     struct  overlap_ent_node   *overlap_ptr,   /* pointer to node */
                 *overlap_ent_node_alloc( ); /* pointer to newly */
                                          /* allocated node   */
     char temp_str[NUMDIGIT + 1];
     char name_str[ENLength + 1];
     int  done_flag;

   /* this function allocates a new subtype with one or more entity node */
   /* and returns a pointer to it                                         */

#ifdef EnExFlag
      printf("Enter rd_overlap_ent_node");
#endif

     /* get new node and initialize pointers */

     overlap_ptr = overlap_ent_node_alloc( );
     readstr(fid, name_str);

     done_flag = FALSE;
     while (done_flag == FALSE)
        if (strcmp(name_str, ent_ptr->en_name) == 0)
          {
           overlap_ptr->oen_name = ent_ptr;
           done_flag = TRUE;
          }
        else
          {
           ent_ptr = ent_ptr->en_next_ent;
           if (ent_ptr == NULL) done_flag = TRUE;
           }

     overlap_ptr->oen_next_name = NULL;

#ifdef EnExFlag
      printf("Exit rd_overlap_ent_node");
#endif

     return(overlap_ptr);
   } /* end overlap_ent_node */
```

```
static struct overlap_sub_node   *rd_overlap_sub_node(fid,gen_ptr)
    FILE  *fid;
    struct  gen_sub_node  *gen_ptr;
  {
    struct overlap_sub_node    *overlapsub_ptr,  /* pointer to termimal */
                                        /* subtype nodes        */
                *overlap_sub_node_alloc();  /* pointer to newly     */
                                        /* allocated node       */
    char temp_str[NUMDIGIT + 1];
    char name_str[ENLength + 1];
    int  done_flag;

  /* this function allocates a new terminal subtype node and returns a */
  /* pointer to it                                                     */

#ifdef EnExFlag
      printf("Enter rd_overlapsub_node");
#endif

    /* get new terminal subtype node and initialize pointers */

    overlapsub_ptr = overlap_sub_node_alloc();
    readstr(fid, name_str);

    done_flag = FALSE;
    while (done_flag == FALSE)
       if (strcmp(name_str, gen_ptr->gsn_name) == 0)
         {
          overlapsub_ptr->osn_name = gen_ptr;
          done_flag = TRUE;
         }
       else
         {
          gen_ptr = gen_ptr->gsn_next_genptr;
          if (gen_ptr == NULL) done_flag = TRUE;
          }

    overlapsub_ptr->osn_next_name = NULL;

#ifdef EnExFlag
      printf("Exit rd_overlapsub_node");
#endif

    return(overlapsub_ptr);
  }  /* end rd_overlapsub_node */


 static struct sub_non_node    *rd_sub_non_node(fid)
    FILE  *fid;
  {
    struct sub_non_node    *subnon_ptr,  /* pointer to subtype nonentity */
                                    /* node                      */
           *sub_non_node_alloc( );  /* pointer to newly allocated   */
                                    /* nonentity node               */
    char temp_str[NUMDIGIT + 1];
```

```
  /* this function allocates a new subtype nonentity node and returns a */
  /* pointer to it                                                      */

#ifdef EnExFlag
      printf("Enter rd_sub_non_node");
#endif

    /* get new subtype nonentity node and initialize pointers */

    subnon_ptr = sub_non_node_alloc( );
    subnon_ptr->snn_value = NULL;
    subnon_ptr->snn_next_node = NULL;

    /* now the subtype nonentity node is filled in by reading the file */

    readstr(fid,subnon_ptr->snn_name);
    readstr(fid,temp_str);
    subnon_ptr->snn_type = temp_str[0];
    readstr(fid,temp_str);
    subnon_ptr->snn_total_length = str_to_num(temp_str);
    readstr(fid,temp_str);
    subnon_ptr->snn_range = str_to_num(temp_str);
    readstr(fid,temp_str);
    subnon_ptr->snn_num_values = str_to_num(temp_str);

#ifdef EnExFlag
      printf("Exit rd_sub_non_node");
#endif

    return(subnon_ptr);
  } /* end rd_sub_non_node */


 static struct der_non_node    *rd_der_non_node(fid)
   FILE *fid;
  {
   struct  der_non_node   *dernon_ptr,  /* pointer to derived type */
                                        /* nonentity node          */
           *der_non_node_alloc( );  /* pointer to newly allocated */
                                        /* derived type            */
   char temp_str[NUMDIGIT + 1];

  /* this function allocates a new derived type nonentity node and returns */
  /* a pointer to it                                                       */

#ifdef EnExFlag
      printf("Enter rd_der_non_node");
#endif

    /* get new derived type nonentity node and initialize pointers */

    dernon_ptr = der_non_node_alloc( );
    dernon_ptr->dnn_value = NULL;
    dernon_ptr->dnn_next_node = NULL;

    /* now the derived type nonentity node is filled */
```

```
    readstr(fid,dernon_ptr->dnn_name);
    readstr(fid,temp_str);
    dernon_ptr->dnn_type = temp_str[0];
    readstr(fid,temp_str);
    dernon_ptr->dnn_total_length = str_to_num(temp_str);
    readstr(fid,temp_str);
    dernon_ptr->dnn_range = str_to_num(temp_str);
    readstr(fid,temp_str);
    dernon_ptr->dnn_num_values = str_to_num(temp_str);

#ifdef EnExFlag
     printf("Exit rd_der_non_node");
#endif

    return(dernon_ptr);
  } /* end rd_dernon_node */
```

APPENDIX C

THE LIL MODULE

```
    #include <stdio.h>
    #include "licommdata.def"
    #include "struct.def"
    #include "flags.def"
    #include "dap.ext"
    #include "lil.dcl"

language_interface_layer()
  /* This proc allows the user to interface with the system.  */
  /* Input and output: user DAPLEX requests                    */

{
  int    num;
  int    stop;       /* boolean flag */

#Ifdef EnExFlag
  printf ("Enter language_interface_layer0); #endif

  dap_init();

  /* initialize several ptrs to different parts of the user structure */
  /* for ease of access                                               */
  dap_info_ptr  = &(cuser_dap_ptr->ui_li_type.li_dap);
  tran_info_ptr = &(dap_info_ptr->dpi_dml_tran);
  first_req_ptr = &(tran_info_ptr->ti_first_req);
  curr_req_ptr  = &(tran_info_ptr->ti_curr_req);
  stop = FALSE;
  while (stop == FALSE)
    {
     /* allow user choice of several processing operations */
     printf ("0nter type of operation desired0);
     printf ("(l) - load new database0);
     printf ("(p) - process existing database0);
     printf ("(x) - return to the operating system0);
     dap_info_ptr->dap_answer = get_ans(&num);

     switch (dap_info_ptr->dap_answer)
```

```
        {
          case 'l': /* user desires to load a new database */
                    load_new();
                    break;
          case 'p': /* user desires to process an existing database */
                    process_old();
                    break;
          case 'x': /* user desires to exit to the operating system */
                    /* database list must be saved back to a file   */
                    stop = TRUE;
                    break;
          default:  /* user did not select a valid choice from the menu */
                    printf ("0rror - invalid operation selected0);
                    printf ("Please pick again0);
                    break;
        } /* end switch */

     /* return to main menu */

   } /* end while */

#fdef EnExFlag
   printf ("Exit language_interface_layer0); #endif

} /* end language_interface_layer */

dap_init() {

#fdef EnExFlag
   printf ("Enter dap_init0); #endif

#fdef EnExFlag
   printf ("Exit dap_init0); #endif

} /* end dap_init */

load_new()
  /*  This proc accomplishes the following:                      */
  /*  (1) determines if the new database name already exists,    */
  /*  (2) adds a new header node to the list of schemas,         */
  /*  (3) determines the user input mode (file/terminal),        */
  /*  (4) reads the user input and forwards it to the parser, and */
  /*  (5) calls the routine that builds the template/descriptor files */
```

```
{
   int   num;
   int   stop;          /* boolean flag */
   int   more_input;    /* boolean flag */
   int   proceed;       /* boolean flag */
   struct ent_dbid_node *db_list_ptr,               /* ptr to the current db */
                        *new_ptr,                /* ptr to a new db structure */
                        *ent_dbid_node_alloc(); /* ptr to allocated db */

#fdef EnExFlag
  printf ("Enter load_new0); #endif

  /* prompt user for name of new database */
  printf ("[7;7m0nter name of database ---->[0;0m ");
  readstr (stdin, dap_info_ptr->dpi_curr_db.cdi_dbname);
  to_caps (dap_info_ptr->dpi_curr_db.cdi_dbname);
  db_list_ptr = dbs_dap_head_ptr.dn_dap;
  stop = FALSE;
  while (stop == FALSE)
   {
     /* determine if new database name already exists */
     /* by traversing list of entity-relation db schemas */
     if ((strcmp(db_list_ptr->edn_name,
               dap_info_ptr->dpi_curr_db.cdi_dbname))== 0)
      {
        printf ("0rror - db name already exists0);
        printf ("[7;7mPlease reenter db name ---->[0;0m ");
        readstr  (stdin,  dap_info_ptr->dpi_curr_db.cdi_dbname);          to_caps  (dap_info_ptr-
>dpi_curr_db.cdi_dbname);
        db_list_ptr = dbs_dap_head_ptr.dn_dap;
      } /* end if */
     else       /* check for last database of the list */       if (db_list_ptr->edn_next_db  ==
NULL)        stop = TRUE;       else
        /* increment to next database */
        db_list_ptr = db_list_ptr->edn_next_db;
   } /* end while */

  /* continue - user input a valid 'new' database name */
  /* add new header node to the list of schemas and fill-in db name */
  /* append new header node to db_list & init relevent user stucture ptrs */
```

```
new_ptr = ent_dbid_node_alloc();
strcpy (new_ptr->edn_name, dap_info_ptr->dpi_curr_db.cdi_dbname);
/* new_ptr->dpidn_num_set = 0; */
/* new_ptr->dpidn_num_rec = 0; */
/* new_ptr->dpidn_first_set = NULL; */
/* new_ptr->dpidn_first_rec = NULL; */
/* new_ptr->dpidn_next_db = NULL; */
db_list_ptr->edn_next_db = new_ptr;
dap_info_ptr->dpi_curr_db.cdi_db.dn_dap = new_ptr;
dap_info_ptr->dpi_curr_db.cdi_attr.an_dattr_ptr = NULL;

/* check for user's mode of input */
more_input = TRUE;
while (more_input == TRUE)
  {
    /* determine user's mode of input */
    printf ("0nter mode of input desired0);
    printf ("(f) - read in database description from a file0);
    printf ("(x) - return to the to main menu0);
    dap_info_ptr->dap_answer = get_ans(&num);

    switch (dap_info_ptr->dap_answer)
      {
        case 'f': /* user input is from a file */
                read_transaction_file();                if (dap_info_ptr->dap_error != ErrReadFile)
                {                /* file contains transactions */                /* dbd stands for
database   description   */                dbd_to_KMS();                free_requests();
                if (dap_info_ptr->dap_error != ErrCreateDB)                {
/* no syntax errors in creates */
                        build_ddl_files();
                      Kernel_Controller();                } /* end if */                } /* end if */
                break;
        case 'x': /* exit back to LIL */
                more_input = FALSE;
                break;
        default:  /* user did not select a valid choice from the menu */
                printf ("0rror - invalid input mode selected0);
                printf ("Please pick again0);
                break;
```

```
      } /* end switch */

     if (dap_info_ptr->dap_error == ErrCreateDB)        /* errors in creates so exit this loop */
      more_input = FALSE;
     dap_info_ptr->dap_error = NOErr;
    } /* end while */

#Ifdef EnExFlag
  printf ("Exit load_new0); #endif

} /* end load_new */

process_old()
  /*  This proc accomplishes the following:                 */
  /*  (1) determines if the database name already exists,    */
  /*  (2) determines the user input mode (file/terminal),    */
  /*  (3) reads the user input and forwards it to the parser */

{
   int    found, more_input;          /* boolean flags */
   int    num;
   struct ent_dbid_node *db_list_ptr; /* ptr to the current database */


#Ifdef EnExFlag
  printf ("Enter process_old0); #endif

  /* prompt user for name of existing database */
  printf ("[7;7m0nter name of database ---->[0;0m ");
  readstr (stdin, dap_info_ptr->dpi_curr_db.cdi_dbname);
  to_caps (dap_info_ptr->dpi_curr_db.cdi_dbname);
  db_list_ptr = dbs_dap_head_ptr.dn_dap;

  found = FALSE;
  while (found == FALSE)
   {
     /* determine if database name does exist      */
     /* by traversing list of entity-relation schemas */
     if (strcmp(dap_info_ptr->dpi_curr_db.cdi_dbname,db_list_ptr->edn_name)== 0)
      found = TRUE;
     else
      {
```

```
db_list_ptr = db_list_ptr->edn_next_db;
/* error condition causes end of list('NULL') to be reached */
if (db_list_ptr == NULL)
  {
    printf ("0rror - db name does not exist0);
    printf ("[7;7mPlease reenter valid db name ---->[0;0m ");
    readstr     (stdin,     dap_info_ptr->dpi_curr_db.cdi_dbname);                    to_caps
(dap_info_ptr->dpi_curr_db.cdi_dbname);
    db_list_ptr = dbs_dap_head_ptr.dn_dap;
  } /* end if */

} /* end else */

} /* end while */

/* continue - user input a valid existing database name */
/* determine user's mode of input */
more_input = TRUE;
while (more_input == TRUE)
  {
    printf ("0nter mode of input desired0);
    printf ("(f) - read in a group of DAPLEX requests from a file0);
    printf ("(t) - read in DAPLEX requests from the terminal0);
    printf ("(x) - return to the previous menu0);
    dap_info_ptr->dap_answer = get_ans(&num);

    switch (dap_info_ptr->dap_answer)
      {
        case 'f': /* user input is from a file */
          read_transaction_file();          dapreqs_to_KMS();
         free_requests();          break;
        case 't': /* user input is from the terminal */
          read_terminal();
          dapreqs_to_KMS();          free_requests();          break;
        case 'x': /* user wishes to return to LIL menu */
          more_input = FALSE;
          break;
        default: /* user did not select a valid choice from the menu */
          printf ("0rror - invalid input mode selected0);
          printf ("Please pick again0);          break;
```

```
    } /* end switch */
  } /* end while */

#fdef EnExFlag
  printf ("Exit process_old0); #endif

} /* end process_old */
```

APPENDIX D

THE KMS MODULE

```
/* Last Date Modified: 12 Nov 85 / ja / building db description ops */

{

    #include <stdio.h>
    #include "licommdata.def"
    #include "struct.def"
    #include "dap.ext"
    #include "flags.def"

int creating = FALSE; int serror; int in; int in1; int in2; int in3; int in4; int in5; int add;
int present; int there; int i, j; int move, nmove; int b; int nsub,esub; int curr_op; int
check_ids; int dummy, dummy2; int ada_expression;

char temp_str[NUMDIGIT + 1]; char db[DBNlength + 1]; char temp_value[ENlength + 1]; char
temp[ENlength + 1]; char type_name_id[ENlength + 1];

struct ent_dbid_node *db_ptr; struct ent_non_node *non_ent_ptr1,
        *non_ent_ptr2; struct ent_value *entval_ptr1,
        *entval_ptr2; struct sub_non_node *subnon_ptr1,
        *subnon_ptr2; struct der_non_node *dernon_ptr1,
        *dernon_ptr2; struct ent_node *ent_ptr1,
        *ent_ptr2;
        *new_ent_ptr; struct gen_sub_node *gen_ptr1,
        *gen_ptr2;
        *new_gen_ptr; struct overlap_ent_node *overlapent_ptr1,
         *overlapent_ptr2; struct overlap_sub_node *overlapsub_ptr1,
         *overlapsub_ptr2; struct function_node *func_ptr1,
         *func_ptr2; struct dap_kms_info *kms_ptr; struct dap_kms_info
*dap_kms_info_alloc(); struct ident_list *id_ptr,
         *temp_ptr,
         *new_temp_ptr,
         *new_id_ptr; struct ent_non_node *dap_ent_non_node_alloc(); struct
ent_node *dap_ent_node_alloc(); struct function_node *dap_func_node_alloc(); struct
sub_non_node *dap_sub_non_node_alloc(); struct der_non_node
```

```
*dap_der_non_node_alloc(); struct ent_value        *dap_ent_value_alloc(); struct dap_create_list
*create_list1,
                  *create_list2,
                  *dap_create_list_alloc();

%}

%union
    {
     char  str[20];
    }

%token  DATABASE %token  ENTITY %token  OVERLAP %token  TEMPORARY %token
TRUE %token  FALSE %token  END %token  IS %token  WITH %token  WITHIN %token
UNIQUE  %token  TYPE  %token  SUBTYPE  %token  NEW  %token  EMPTY  %token
CREATE %token  CONSTANT %token  AND %token  OR %token  XOR %token  THEN
%token  ELSE %token  FOR %token  EACH %token  DELTA %token  NULL %token  WITH-
NULL %token  WITHOUTNULL %token  SET %token  IMAGE %token  POS %token  VALUE
%token  VAL

%token  <str>  IDENTIFIER %token  <str>  NUMERIC_LITERAL %token  <str>  STRING
%token  <str>  CHARACTER_STRING %token  <str>  LITERAL_STRING %token  <str>
FLOAT  %token  <str>  INTEGER  %token  <str>  BOOLEAN  %token  <str>  RANGE
%token  <str>  DIGITS  %token  <str>  ELIPSES  %token  <str>  COLON  %token  <str>
SEMICOLON %token  <str>  DOT %token  <str>  COMMA %token  <str>  ASSIGN %token
<str>  LP %token  <str>  RP %token  <str>  HYPHEN %token  <str>  IMPLY

%token  <str>  subtype_indicator %token  <str>  subtype_indication

%start statement

%%

statement: ddl_statement
        {
         YYACCEPT;
        }
        ;

ddl_statement: database_specification
```

```
            {
             proc_free_id_list();
            }
            ;

database_specification: TEMPORARY database_definition
                | database_definition
                ;

database_definition: DATABASE
               { #ifdef HYacFlag
        printf("Database in database_definition recognized0); #endif
                check_ids = TRUE;
                creating = TRUE;
               }
               visible_part end_database
               ;

end_database: end_module
          ;

end_module: END
        | END
          {
           check_ids = FALSE;
          }
         name_id
         {
          db_ptr = cuser_dap_ptr->ui_li_type.li_dap.dpi_curr_db.cdi_db.dn_dap;
          strcpy(db,db_ptr->edn_name);
          if(strcmp(temp_value, db_ptr->edn_name) != FALSE)
           {
            serror = 0;
            proc_eval_error(serror);
            YYACCEPT;
           }
         }
        ;

visible_part: name_id        /* ='s [A-Z][A-Z]  in LEX */
```

```
/* the ent_dbid_node is located and compared for correctness */

  {
   db_ptr = cuser_dap_ptr->ui_li_type.li_dap.dpi_curr_db.cdi_db.dn_dap;
   strcpy(db,db_ptr->edn_name);
   if (strcmp(db_ptr->edn_name, temp_value) != FALSE)
     {
      serror = 0;
      proc_eval_error(serror);
      YYACCEPT;
     }
   }

IS declarative_item_list
  ;

declarative_item_list: declarative_item
          | declarative_item_list declarative_item
          ;

declarative_item: declaration
          | consistency_rule
          ;

consistency_rule: overlap_rule
          |  uniqueness_rule
          ;

overlap_rule: OVERLAP

  /* the types are checked to insure that they are terminal subtypes */

   {
    serror = 14;
    check_ids = FALSE;
    curr_op = Overlap;
   }
   name1_list
   {
    kms_ptr->dki_overfirst_ptr = kms_ptr->dki_temp_ptr;
    kms_ptr->dki_temp_ptr = NULL;
    ov_ptr = kms_ptr->dki_overfirst_ptr;
```

```
in = FALSE;
new_gen_ptr = db_ptr->edn_subptr;
gen_ptr = new_gen_ptr;
while (ov_ptr != NULL)
  {
   while ((gen_ptr != NULL) && (in != TRUE))
     {
      if (strcmp(ov_ptr->il_name, gen_ptr->gsn_name) == FALSE)
        {
         in = TRUE;
        }
      else
        {
         new_gen_ptr = gen_ptr->gsn_next_genptr;
         gen_ptr = new_gen_ptr;
        }
     }
   if (in == TRUE)
     {
      new_ov_ptr = ov_ptr->il_next;
      ov_ptr = new_ov_ptr;
      new_gen_ptr = db_ptr->edn_subptr;
      gen_ptr = new_gen_ptr;
      in = FALSE;
     }
   else
     {
      proc_eval_error(serror);
     }
  }
}
WITH name1_list SEMICOLON
{
 in = FALSE;
 new_temp_ptr = kms_ptr->dki_temp_ptr;
 temp_ptr = new_temp_ptr;
 new_gen_ptr = db_ptr->edn_subptr;
 gen_ptr = new_gen_ptr;
```

```
new_overlapsub_ptr = gen_ptr->gsn_subptr;
overlapsub_ptr = new_overlapsub_ptr;
while (temp_ptr != NULL)
  {
   while ((gen_ptr != NULL) && (in == FALSE))
     {
      if (strcmp(gen_ptr->gsn_name, temp_ptr->il_name) == FALSE)
       {
        in = TRUE;
        new_ov_ptr = kms_ptr->dki_overfirst_ptr;
        ov_ptr = new_ov_ptr;
        while (ov_ptr != NULL)
          {
           if (overlapsub_ptr == NULL)
            {
             gen_ptr->gsn_subptr = dap_overlap_sub_node_alloc();
             new_overlapsub_ptr = gen_ptr->gsn_subptr;
             overlapsub_ptr = new_overlapsub_ptr;
             ov_ptr->il_name = overlapsub_ptr;
             overlapsub_ptr->osn_name = ov_ptr->il_name;
             overlapsub_ptr->oen_next_name = NULL;
            }
           else
            {
             while (overlapsub_ptr->oen_name != NULL)
               overlapsub_ptr = overlapsub_ptr->oen_next_name;
             overlapsub_ptr->oen_next_name = dap_overlap_sub_node_alloc();
             new_overlapsub_ptr = overlapsub_ptr->oen_next_name;
             overlapsub_ptr = new_overlapsub_ptr;
             ov_ptr->il_name = overlapsub_ptr;
             overlapsub_ptr->osn_name = ov_ptr->il_name;
             overlapsub_ptr->oen_next_name = NULL;
            }
           new_ov_ptr = ov_ptr->il_next;
           ov_ptr = new_ov_ptr;
          }
       }
      else
```

```
      {
       new_gen_ptr = gen_ptr->gsn_next_ptr;
       gen_ptr = new_gen_ptr;
      }
     }
    if ((gen_ptr == NULL) && (in == FALSE))
     {
      proc_eval_error(serror);
     }
    else
     {
      new_temp_ptr = temp_ptr->il_next;
      temp_ptr = new_temp_ptr;
      new_gen_ptr = db_ptr->edn_subptr;
      gen_ptr = new_gen_ptr;
      in == FALSE;
     }
    }
   check_ids = TRUE;
  }
 ;

uniqueness_rule: UNIQUE identifier_list WITHIN name1 SEMICOLON
    {
     curr_op = Unique;
     check_ids = FALSE;
     serror = 13;

     /* to create temp_list must be entity type or subtype */

     in = FALSE;
     new_ent_node_ptr = db_ptr->edn_entity;
     ent_node_ptr = new_ent_node_ptr;
     new_gen_ptr = db_ptr->edn_subptr;
     gen_ptr = new_gen_ptr;
     while (ent_node_ptr != NULL)
      {
       if (strcmp(ent_node_ptr->en_name, temp_value) == FALSE)
        {
```

```
/* if the temp_value is found  in the function */
/* node the unique field in the function node  */
/* is initialized to true, else an error mess- */
/* is initiated                                 */

in = TRUE;
there = FALSE;
new_func_ptr = ent_node_ptr->en_ftnptr;
func_ptr = new_func_ptr;
new_temp_ptr = kms_ptr->dki_temp_ptr;
temp_ptr = new_temp_ptr;
while (temp_ptr != NULL)
  {
   while (func_ptr != NULL)
     {
      if(strcmp(temp_ptr->il_name, func_ptr->fn_name) == FALSE)
       {
        there = TRUE;
        func_ptr->fn_unique = TRUE;
       }
      else
       {
        new_func_ptr = func_ptr->fn_next_fntptr;
        func_ptr = new_func_ptr;
       }
     }
   if (there == TRUE)
     {
      new_temp_ptr = temp_ptr->il_next;
      temp_ptr = new_temp_ptr;
      new_func_ptr = ent_node_ptr->en_ftnptr;
      func_ptr = new_func_ptr;
      there = FALSE;
     }
   else
     {
      proc_eval_error(serror);
     }
  }
```

```
        }
       else
        {
         new_ent_node_ptr = ent_node_ptr->en_next_ent;
         ent_node_ptr = new_ent_node_ptr;
        }
      }
   if (in == FALSE)
    {

  /* The temp_value is compared to each value in the */
  /* gen_sub_node. If the value is not there, an error */
  /* message is initiated.                             */

    while (gen_ptr != NULL)
      {
       if (strcmp(gen_ptr->gsn_name, temp_value) == FALSE)
        {
         in = TRUE;
         there = FALSE;
         new_func_ptr = gen_ptr->gsn_ftnptr;
         func_ptr = new_func_ptr;
         new_temp_ptr = kms_ptr->dki_temp_ptr;
         temp_ptr = new_temp_ptr;
         while (temp_ptr != NULL)
           {
            while (func_ptr != NULL)
              {
               if(strcmp(temp_ptr->il_name, func_ptr->fn_name) == FALSE)
                {
                 there = TRUE;
                 func_ptr->fn_unique = TRUE;
                }
               else
                {
                 new_func_ptr = func_ptr->fn_next_fntptr;
                 func_ptr = new_func_ptr;
                }
              }
```

```
              if (there == TRUE)
                {
                 new_temp_ptr = temp_ptr->il_next;
                 temp_ptr = new_temp_ptr;
                 new_func_ptr = gen_node_ptr->gsn_ftnptr;
                 func_ptr = new_func_ptr;
                 there = FALSE;
                }
              else
                {
                 proc_eval_error(serror);
                }
             }
           }
          else
           {
            new_gen_ptr = gen_ptr->gsn_next_genptr;
            gen_ptr = new_gen_ptr;
           }
         }
        }
       if (in == FALSE)
        proc_eval_error(serror);
       check_ids = TRUE;
      }
    ;

declaration: number_declaration
       | type_declaration
       | subtype_declaration
       ;

number_declaration:
        {
         serror = 1;
         check_ids = TRUE;
        }
       identifier_list COLON CONSTANT ASSIGN simple_const SEMICOLON
        {
```

```
        temp_ptr = kms_ptr->dki_temp_ptr;
        while (temp_ptr != NULL)
          {
            /* At this point ent_non_node's are filled with the */
            /* information previously allocated in the kms info */
            /* structure.  The amount of nodes is dependent on  */
            /* the amount of names in the temp structure.       */

            ent_non_ptr1 = dap_ent_non_node_alloc();
            strcpy(ent_non_ptr1->enn_name, temp_ptr->il_name);
            ent_non_ptr1->enn_type = kms_ptr->dki_ent_non.enn_type ;
            ent_non_ptr1->enn_total_length =
                          kms_ptr->dki_ent_non.enn_total_length ;
            ent_non_ptr1->enn_range = kms_ptr->dki_ent_non.enn_range ;
            ent_non_ptr1->enn_num_values =
                          kms_ptr->dki_ent_non.enn_num_values ;
            ent_non_ptr1->enn_value = kms_ptr->dki_ent_non.enn_value ;
            kms_ptr->dki_ent_non.enn_value = NULL;
            ent_non_ptr1->enn_constant =
                          kms_ptr->dki_ent_non.enn_constant ;
            ent_non_ptr1->enn_next_node = NULL;

            ent_non_ptr2 = db_ptr->edn_nonentity;

            if (ent_non_ptr2 == NULL)
              db_ptr->edn_nonentity = ent_non_ptr1;
            else
              {
                while (ent_non_ptr2->enn_next_node != NULL)
                  ent_non_ptr2 = ent_non_ptr2->enn_next_node;
                ent_non_ptr2->enn_next_node = ent_non_ptr1;
              }
            ent_non_ptr1 = NULL;
            temp_ptr = temp_ptr->il_next;
          }
        check_ids = FALSE;
      }
    ;

simple_const:  INTEGER   /* dap_kms_info structures are built for subsequent */
```

```
        /* nonentity node insertion into the schema        */
    {
      kms_ptr->dki_ent_non.enn_type = 'i';
      kms_ptr->dki_ent_non.enn_total_length = INTLength;
      kms_ptr->dki_ent_non.enn_range = FALSE;
      kms_ptr->dki_ent_non.enn_num_values = 1;
      kms_ptr->dki_ent_non.enn_constant = TRUE;
      kms_ptr->dki_ent_non.enn_value = dap_ent_value_alloc();
      kms_ptr->dki_ent_non.enn_value->ev_value =
                var_str_alloc( strlen($1) + 1 );
      strcpy(kms_ptr->dki_ent_non.enn_value->ev_value, $1);
      kms_ptr->dki_ent_non.enn_value->ev_next_value = NULL;
    }
  | FLOAT
    {
      kms_ptr->dki_ent_non.enn_type = 'f';
      kms_ptr->dki_ent_non.enn_total_length = FLTLength;
      kms_ptr->dki_ent_non.enn_range = FALSE;
      kms_ptr->dki_ent_non.enn_num_values = 1;
      kms_ptr->dki_ent_non.enn_constant = TRUE;
      kms_ptr->dki_ent_non.enn_value = dap_ent_value_alloc();
      kms_ptr->dki_ent_non.enn_value->ev_value =
                var_str_alloc( strlen($1) + 1 );
      strcpy(kms_ptr->dki_ent_non.enn_value->ev_value, $1);
      kms_ptr->dki_ent_non.enn_value->ev_next_value = NULL;
    }
    ;

type_declaration: TYPE
          {
            curr_op = TypeIs;
            check_ids = FALSE;
            serror = 2;
          }
          name_id
          {
            strcpy(temp_name_id, temp_value);
            check_ids = TRUE;
```

```
  serror = 9;
}
IS type_definition SEMICOLON

/* the following switch statement allocates a nonentity */
/* derived, or entity node to the schema dependent upon */
/* the value of curr_op                              */

{
  curr_op = CheckIds;
  switch(curr_op)
   {
    case NonEnt:
         ent_non_ptr1 = dap_ent_non_node_alloc();
         strcpy( ent_non_ptr1->enn_name, temp_name_id );
         ent_non_ptr1->enn_type = kms_ptr->dki_ent_non.enn_type ;
         ent_non_ptr1->enn_total_length =
                   kms_ptr->dki_ent_non.enn_total_length ;
         ent_non_ptr1->enn_range = kms_ptr->dki_ent_non.enn_range ;
         ent_non_ptr1->enn_num_values =
                   kms_ptr->dki_ent_non.enn_num_values ;
         ent_non_ptr1->enn_value = kms_ptr->dki_ent_non.enn_value ;
         kms_ptr->dki_ent_non.enn_value = NULL;
         ent_non_ptr1->enn_constant =
                   kms_ptr->dki_ent_non.enn_constant ;
         ent_non_ptr1->enn_next_node = NULL;

         ent_non_ptr2 = db_ptr->edn_nonentity;

         if (ent_non_ptr2 == NULL)
          db_ptr->edn_nonentity = ent_non_ptr1;
         else
          {
           while (ent_non_ptr2->enn_next_node != NULL)
             ent_non_ptr2 = ent_non_ptr2->enn_next_node;
           ent_non_ptr2->enn_next_node = ent_non_ptr1;
          }
         ent_non_ptr1 = NULL;
         break;

    case Derived:
```

```
                dernon_ptr1 = dap_der_non_node_alloc();
                strcpy( dernon_ptr1->dnn_name, temp_name_id );
                dernon_ptr1->dnn_type = kms_ptr->dki_der_non.dnn_type ;
                dernon_ptr1->dnn_total_length =
                        kms_ptr->dki_der_non.dnn_total_length ;
                dernon_ptr1->dnn_range = kms_ptr->dki_der_non.dnn_range ;
                dernon_ptr1->dnn_num_values =
                        kms_ptr->dki_der_non.dnn_num_values ;
                dernon_ptr1->dnn_value = kms_ptr->dki_der_non.dnn_value ;
                kms_ptr->dki_der_non.dnn_value = NULL;
                dernon_ptr1->dnn_next_node = NULL;
                dernon_ptr2 = db_ptr->edn_nonderptr;
                if (dernon_ptr2 == NULL)
                 db_ptr->edn_nonderptr = dernon_ptr1;
                else
                 {
                  while (dernon_ptr2->dnn_next_node != NULL)
                    dernon_ptr2 = dernon_ptr2->dnn_next_node;
                  dernon_ptr2->dnn_next_node = dernon_ptr1;
                 }
                dernon_ptr1 = NULL;
                break;

            case Entity:
***         /* check if name_id is on the ent list of the schema */
            }
        }

        | incomplete_type_declaration
        ;

name_id: IDENTIFIER

    /* this rule assigns IDENTIFIER to the variable temp_value */
    /* and inserts it into the id structure of dap_kms_info for */
    /* subsequent comparisons of uniqueness                     */

    {
     strcpy(temp_value,$1);
     id_ptr = kms_ptr->dki_id_ptr;
     if (id_ptr == NULL)
```

```
 {
  kms_ptr->dki_id_ptr = dap_ident_list_alloc();
  id_ptr = kms_ptr->dki_id_ptr;
 }
else
 {
  nmove = FALSE;
  new_id_ptr = id_ptr;
  while(id_ptr != NULL)
    {
     if (strcmp(id_ptr->il_name, temp_value) == FALSE)
      {
       nmove = TRUE;
      }
     else
      {
       new_id_ptr = id_ptr;
       id_ptr = id_ptr->il_next;
      }
    }
  if((nmove == FALSE) && (curr_op == CheckIds))
   {
    new_id_ptr->il_next = dap_ident_list_alloc();
    new_id_ptr = new_id_ptr->il_next;
    strcpy(new_id_ptr->il_name, temp_value);
    new_id_ptr->il_next = NULL;
   }
  else
   {
    if((nmove == TRUE) && (curr_op = CheckIds))
      proc_eval_error(serror);
   }

 }
}
;

incomplete_type_declaration: TYPE
                          /* entity */
```

```
{
  check_ids = TRUE;
  serror = 3;
}
name_id SEMICOLON

/* At this point a check is made to see if the */
/* IDENTIFIER is already in the ent_node. If it */
/* is, an error is produced.  If it is not, it is */
/* added to the schema.                          */

 {
  in = FALSE;
  ent_node_ptr1 = db_ptr->edn_entity;
  ent_node_ptr2 = ent_node_ptr1;
  while (ent_node_ptr2 != NULL)
    {
     if (strcmp(ent_node_ptr2->en_name, temp_value) == FALSE)
      {
       proc_eval_error(serror);
       in = TRUE;
      }
     else
       ent_node_ptr1 = ent_node_ptr1->en_next_ent;
       ent_node_ptr2 = ent_node_ptr1;
    }
  if (in == FALSE)
   {
    ent_node_ptr2 = dap_ent_node_alloc();
    strcpy(ent_node_ptr2->en_name, temp_value);
    ent_node_ptr2->en_num_funct = 0;
    ent_node_ptr2->en_terminal = FALSE;
    ent_node_ptr2->en_ftnptr = NULL;
    ent_node_ptr2->en_next_ent = NULL;

    ent_node_ptr1 = db_ptr->edn_entity;
    ent_node_ptr2 = ent_node_ptr1;
    if (ent_node_ptr2 == NULL)
     {
```

```
            db_ptr->edn_entity = ent_node_ptr2;
            ent_node_ptr2 = NULL;
           }
          else
           {
            while (ent_node_ptr2->en_next_ent != NULL)
              ent_node_ptr1 = ent_node_ptr1->en_next_ent;
              ent_node_ptr2 = ent_node_ptr1;
            ent_node_ptr1->en_next_ent = ent_node_ptr2;
            ent_node_ptr2 = NULL;
           }
          }
        check_ids = FALSE;
        }
      ;

type_definition: /* curr_op variables are set for subsequent switch statement */
        /* utilization                                          */

        {
         curr_op = NonEnt;
        }
        enumeration_type_definition
       |
        {
         curr_op = NonEnt;
        }
        integer_type_definition
       |
        {
         curr_op = NonEnt;
        }
        real_type_definition
       |
        {
         curr_op = Derived;
        }
        derived_type_definition
       |
```

```
          {
           curr_op = Entity;
          }
          entity_type_definition
         ;

enumeration_type_definition: LP

                /* enumeration dap_kms_info structures for */
                /* nonentity and function nodes are initialized */

                {
                 check_ids = FALSE;
                 switch(curr_op)
                  {
                   case NonEnt:
                        kms_ptr->dki_ent_non.enn_type = 'e';
                        kms_ptr->dki_ent_non.enn_range = FALSE;
                        kms_ptr->dki_ent_non.enn_num_values = 0;
                        kms_ptr->dki_ent_non.enn_constant = FALSE;
                        break;

                   case Function:
                        kms_ptr->dki_funct.fn_type = 'e';
                        kms_ptr->dki_funct.fn_range = FALSE;
                        kms_ptr->dki_funct.fn_num_value = 0;
                        break;
                  }
                }
                enumeration_literal_list RP
                {
                 check_ids = TRUE;
                }
                ;

enumeration_literal_list: enumeration_literal

              /* the pointers are set for value nodes with */
              /* concurrent incrementation of the number of */
              /* value nodes present in the nonentity and   */
              /* function structures                        */
```

```
{
 switch(curr_op)
  {
   case NonEnt:
        kms_ptr->dki_ent_non.enn_num_values++;
        kms_ptr->dki_ent_non.enn_value = entval_ptr;
        entval_ptr = NULL;
        break;

   case Function:
        kms_ptr->dki_funct.fn_num_values++;
        kms_ptr->dki_funct.fn_value = entval_ptr;
        entval_ptr = NULL;
        break;
  }
}

| enumeration_literal_list COMMA enumeration_literal

{
 switch(curr_op)
  {
   case NonEnt:
        kms_ptr->dki_ent_non.enn_num_values++;
        entval_ptr2 = kms_ptr->dki_ent_non.enn_value;
        while (entval_ptr2->ev_next_value != NULL)
          entval_ptr2 = entval_ptr2->ev_next_value;
        entval_ptr2 = entval_ptr;
        entval_ptr = NULL;
        break;

   case Function:
        kms_ptr->dki_funct.fn_num_value++;
        entval_ptr2 = kms_ptr->dki_funct.fn_value;
        while (entval_ptr2->ev_next_value != NULL)
          entval_ptr2 = entval_ptr2->ev_next_value;
        entval_ptr2 = entval_ptr;
        entval_ptr = NULL;
        break;
  }
```

```
        }
        ;

enumeration_literal: name_id

        /* ent_value nodes are allocated and the ev_value */
        /* pointer is set the the appropriate IDENTIFIER  */

        {
         switch(curr_op)
          {
           case NonEnt:
                entval_ptr = dap_ent_value_alloc();
                enum_str = var_str_alloc(ENlength + 1);
                strcpy(enum_str, temp_value);
                entval_ptr->ev_value = enum_str;
                entval_ptr->ev_next_value = NULL;
                enum_str = NULL;
                break;

           case Function:
                entval_ptr = dap_ent_value_alloc();
                enum_str = var_str_alloc(ENlength + 1);
                strcpy(enum_str, temp_value);
                entval_ptr->ev_value = enum_str;
                entval_ptr->ev_next_value = NULL;
                enum_str = NULL;
                break;
          }
        }
       | LITERAL_CHARACTER
        {
         switch(curr_op)
          {
           case NonEnt:
                entval_ptr = dap_ent_value_alloc();
                enum_str = var_str_alloc( strlen($1) + 1);
                strcpy(enum_str, $1);
                entval_ptr->ev_value = enum_str;
                entval_ptr->ev_next_value = NULL;
```

```
            enum_str = NULL;
            break;

        case Function:
            entval_ptr = dap_ent_value_alloc();
            enum_str = var_str_alloc( strlen($1) + 1);
            strcpy(enum_str, $1);
            entval_ptr->ev_value = enum_str;
            entval_ptr->ev_next_value = NULL;
            enum_str = NULL;
            break;
        }
      }
      ;

integer_type_definition:

        /* integer type dap_kms_info structures are set for */
        /* subsequent insertion into the schema              */

         {
          check_ids = FALSE;
          switch (curr_op)
           {
            case NonEnt:
                  kms_ptr->dki_ent_non.enn_type = 'i';
                  kms_ptr->dki_ent_non.enn_range = TRUE;
                  kms_ptr->dki_ent_non.enn_num_values = 2;
                  kms_ptr->dki_ent_non.enn_constant = FALSE;
                  break;

            case Derived:
                  kms_ptr->dki_der_non.dnn_type = 'i';
                  kms_ptr->dki_der_non.dnn_range = TRUE;
                  kms_ptr->dki_der_non.dnn_num_values = 2;
                  break;

            case SubNon:
                  kms_ptr->dki_sub_non.snn_type = 'i';
                  kms_ptr->dki_sub_non.snn_range = TRUE;
                  kms_ptr->dki_sub_non.snn_num_values = 2;
```

```
                    break;

                case Function:
                    kms_ptr->dki_funct.fn_type = 'i';
                    kms_ptr->dki_funct.fn_range = TRUE;
                    kms_ptr->dki_funct.fn_num_value = 2;
                    break;
                }
            }
            integer_range
            {
             check_ids = TRUE;
            }
            ;

integer_range: RANGE int_range
        ;

int_range: INTEGER ELIPSES

    /* The kms_infor value nodes are allocated and initialized */
    /* dependent upon the state of curr_op. As can be seen from */
    /* the switch rules which follow, the same sequence must    */
    /* occur for all the allowable types.                      */
      {
      switch (curr_op)
       {
        case NonEnt:
            kms_ptr->dki_ent_non.enn_value = dap_ent_value_alloc();
            kms_ptr->dki_ent_non.enn_value->ev_value = var_str_alloc( strlen($2) + 1);
            strcpy(dki_ent_non.enn_value->ev_value, $2);
            kms_ptr->dki_ent_non.enn_value->ev_next_value = NULL;
            kms_ptr->dki_ent_non.enn_num_values++;
            kms_ptr->dki_ent_non.enn_value = entval_ptr;
            entval_ptr = NULL;
            break;

      case Derived:
            kms_ptr->dki_der_non.dnn_value = dap_ent_value_alloc();
            kms_ptr->dki_der_non.dnn_value->ev_value = var_str_alloc( strlen($2) + 1);
            strcpy(dki_der_non.dnn_value->ev_value, $2);
```

```
        kms_ptr->dki_der_non.dnn_value->ev_next_value = NULL;
        kms_ptr->dki_der_non.dnn_num_values++;
        kms_ptr->dki_der_non.dnn_value = entval_ptr;
        entval_ptr = NULL;
        break;

    case SubNon:
        kms_ptr->dki_sub_non.snn_value = dap_ent_value_alloc();
        kms_ptr->dki_sub_non.snn_value->ev_value = var_str_alloc( strlen($2) + 1);
        strcpy(dki_sub_non.snn_value->ev_value, $2);
        kms_ptr->dki_sub_non.snn_value->ev_next_value = NULL;
        kms_ptr->dki_sub_non.snn_num_values++;
        kms_ptr->dki_sub_non.snn_value = entval_ptr;
        entval_ptr = NULL;
        break;

    case Function:
        kms_ptr->dki_funct.fn_value = dap_ent_value_alloc();
        kms_ptr->dki_funct.fn_value->ev_value = var_str_alloc( strlen($2) + 1);
        strcpy(dki_funct.fn_value->ev_value, $2);
        kms_ptr->dki_funct.fn_value->ev_next_value = NULL;
        kms_ptr->dki_funct.fn_num_value++;
        kms_ptr->dki_funct.fn_value = entval_ptr;
        entval_ptr = NULL;
        break;
     }
  }
INTEGER
  {
   switch (curr_op)
     {
      case NonEnt:
        kms_ptr->dki_ent_non.enn_value = dap_ent_value_alloc();
        kms_ptr->dki_ent_non.enn_value->ev_value = var_str_alloc( strlen($1) + 1);
        strcpy(dki_ent_non.enn_value->ev_value, $1);
        kms_ptr->dki_ent_non.enn_value->ev_next_value = NULL;
        kms_ptr->dki_ent_non.enn_num_values++;
        entval_ptr2 = kms_ptr->dki_ent_non.enn_value;
        entval_ptr2 = entval_ptr;
```

```
            entval_ptr = NULL;
            break;

        case Derived:
            kms_ptr->dki_der_non.dnn_value = dap_ent_value_alloc();
            kms_ptr->dki_der_non.dnn_value->ev_value = var_str_alloc( strlen($1) + 1);
            strcpy(dki_der_non.dnn_value->ev_value, $1);
            kms_ptr->dki_der_non.dnn_value->ev_next_value = NULL;
            kms_ptr->dki_der_non.dnn_num_values++;
            entval_ptr2 = kms_ptr->dki_der_non.dnn_value;
            entval_ptr2 = entval_ptr;
            entval_ptr = NULL;
            break;

        case SubNon:
            kms_ptr->dki_sub_non.snn_value = dap_ent_value_alloc();
            kms_ptr->dki_sub_non.dnn_value->ev_value = var_str_alloc( strlen($1) + 1);
            strcpy(dki_sub_non.snn_value->ev_value, $1);
            kms_ptr->dki_sub_non.snn_value->ev_next_value = NULL;
            kms_ptr->dki_sub_non.snn_num_values++;
            entval_ptr2 = kms_ptr->dki_sub_non.snn_value;
            entval_ptr2 = entval_ptr;
            entval_ptr = NULL;
            break;

        case Function:
            kms_ptr->dki_funct.fn_value = dap_ent_value_alloc();
            kms_ptr->dki_funct.fn_value->ev_value = var_str_alloc( strlen($1) + 1);
            strcpy(dki_funct.fn_value->ev_value, $1);
            kms_ptr->dki_funct.fn_value->ev_next_value = NULL;
            kms_ptr->dki_funct.fn_num_value++;
            entval_ptr2 = kms_ptr->dki_funct.fn_value;
            entval_ptr2 = entval_ptr;
            entval_ptr = NULL;
            break;
         }
        }
        ;

real_type_definition:
```

```
        {
         check_ids = FALSE;
         switch (curr_op)
          {
           case NonEnt:
                kms_ptr->dki_ent_non.enn_type = 'f';
                kms_ptr->dki_ent_non.enn_range = TRUE;
                kms_ptr->dki_ent_non.enn_num_values = 2;
                kms_ptr->dki_ent_non.enn_constant = FALSE;
                break;

           case Derived:
                kms_ptr->dki_der_non.dnn_type = 'f';
                kms_ptr->dki_der_non.dnn_range = TRUE;
                kms_ptr->dki_der_non.dnn_num_values = 2;
                break;

           case SubNon:
                kms_ptr->dki_sub_non.snn_type = 'f';
                kms_ptr->dki_sub_non.snn_range = TRUE;
                kms_ptr->dki_sub_non.snn_num_values = 2;
                break;

           case Function:
                kms_ptr->dki_funct.fn_type = 'f';
                kms_ptr->dki_funct.fn_range = TRUE;
                kms_ptr->dki_funct.fn_num_value = 2;
                break;
          }
        }
        float_range
         {
          check_ids = TRUE;
         }
        ;

float_range: RANGE FLOAT ELIPSES
      {
       switch (curr_op)
        {
```

```
case NonEnt:
    kms_ptr->dki_ent_non.enn_value = dap_ent_value_alloc();
    kms_ptr->dki_ent_non.enn_value->ev_value = var_str_alloc( strlen($2) + 1);
    strcpy(dki_ent_non.enn_value->ev_value, $2);
    kms_ptr->dki_ent_non.enn_value->ev_next_value = NULL;
    kms_ptr->dki_ent_non.enn_num_values++;
    kms_ptr->dki_ent_non.enn_value = entval_ptr;
    entval_ptr = NULL;
    break;

case Derived:
    kms_ptr->dki_der_non.dnn_value = dap_ent_value_alloc();
    kms_ptr->dki_der_non.dnn_value->ev_value = var_str_alloc( strlen($2) + 1);
    strcpy(dki_der_non.dnn_value->ev_value, $2);
    kms_ptr->dki_der_non.dnn_value->ev_next_value = NULL;
    kms_ptr->dki_der_non.dnn_num_values++;
    kms_ptr->dki_der_non.dnn_value = entval_ptr;
    entval_ptr = NULL;
    break;

case SubNon:
    kms_ptr->dki_sub_non.snn_value = dap_ent_value_alloc();
    kms_ptr->dki_sub_non.snn_value->ev_value = var_str_alloc( strlen($2) + 1);
    strcpy(dki_sub_non.snn_value->ev_value, $2);
    kms_ptr->dki_sub_non.snn_value->ev_next_value = NULL;
    kms_ptr->dki_sub_non.snn_num_values++;
    kms_ptr->dki_sub_non.snn_value = entval_ptr;
    entval_ptr = NULL;
    break;

case Function:
    kms_ptr->dki_funct.fn_value = dap_ent_value_alloc();
    kms_ptr->dki_funct.fn_value->ev_value = var_str_alloc( strlen($2) + 1);
    strcpy(dki_funct.fn_value->ev_value, $2);
    kms_ptr->dki_funct.fn_value->ev_next_value = NULL;
    kms_ptr->dki_funct.fn_num_value++;
    kms_ptr->dki_funct.fn_value = entval_ptr;
    entval_ptr = NULL;
    break;
}
```

```
}
FLOAT
{
 switch (curr_op)
  {
   case NonEnt:
        kms_ptr->dki_ent_non.enn_value = dap_ent_value_alloc();
        kms_ptr->dki_ent_non.enn_value->ev_value = var_str_alloc( strlen($1) + 1);
        strcpy(dki_ent_non.enn_value->ev_value, $1);
        kms_ptr->dki_ent_non.enn_value->ev_next_value = NULL;
        kms_ptr->dki_ent_non.enn_num_values++;
        entval_ptr2 = kms_ptr->dki_ent_non.enn_value;
        entval_ptr2 = entval_ptr;
        entval_ptr = NULL;
        break;

   case Derived:
        kms_ptr->dki_der_non.dnn_value = dap_ent_value_alloc();
        kms_ptr->dki_der_non.dnn_value->ev_value = var_str_alloc( strlen($1) + 1);
        strcpy(dki_der_non.dnn_value->ev_value, $1);
        kms_ptr->dki_der_non.dnn_value->ev_next_value = NULL;
        kms_ptr->dki_der_non.dnn_num_values++;
        entval_ptr2 = kms_ptr->dki_der_non.dnn_value;
        entval_ptr2 = entval_ptr;
        entval_ptr = NULL;
        break;

   case Subnon:
        kms_ptr->dki_sub_non.snn_value = dap_ent_value_alloc();
        kms_ptr->dki_sub_non.snn_value->ev_value = var_str_alloc( strlen($1) + 1);
        strcpy(dki_sub_non.snn_value->ev_value, $1);
        kms_ptr->dki_sub_non.snn_value->ev_next_value = NULL;
        kms_ptr->dki_sub_non.snn_num_values++;
        entval_ptr2 = kms_ptr->dki_sub_non.snn_value;
        entval_ptr2 = entval_ptr;
        entval_ptr = NULL;
        break;

   case Function:
        kms_ptr->dki_funct.fn_value = dap_ent_value_alloc();
```

```
            kms_ptr->dki_funct.fn_value->ev_value = var_str_alloc( strlen($1) + 1);
            strcpy(dki_funct.fn_value->ev_value, $1);
            kms_ptr->dki_funct.fn_value->ev_next_value = NULL;
            kms_ptr->dki_funct.fn_num_value++;
            entval_ptr2 = kms_ptr->dki_funct.fn_value;
            entval_ptr2 = entval_ptr;
            entval_ptr = NULL;
            break;
        }
      }
      ;

derived_type_defintion: NEW

            /* the nonentity, subtype nonentity, and derived type */
            /* nonentity nodes are examined to find which con-    */
            /* the current value of IDENTIFIER                    */

             {
              curr_op = Derived;
              check_ids = FALSE;
             }
            name_id
             {
              non_ent_ptr1 = db_ptr->edn_nonentity;
              non_ent_ptr2 = non_ent_ptr1;
              subnon_ptr1 = db_ptr->edn_nonsubptr;
              subnon_ptr2 = subnon_ptr1;
              dernon_ptr1 = db_ptr->edn_nonderptr;
              dernon_ptr2 = dernon_ptr1;
              in1 = FALSE;
              in2 = FALSE;
              in3 = FALSE;
              while ((non_ent_ptr2 != NULL) && (in1 == FALSE))
                {
                 if(strcmp(non_ent_ptr2->enn_name, temp_value) == FALSE)
                  {
                   in1 = TRUE;
                   strcpy(temp, temp_value);
                  }
```

```
        else
         {
          non_ent_ptr1 = non_ent_ptr1->enn_next_node;
          non_ent_ptr2 = non_ent_ptr1;
         }
        }
      while ((subnon_ptr2 != NULL) && (in1 == FALSE) && (in2 == FALSE))
        {
        if(strcmp(subnon_ptr2->snn_name, temp_value) == FALSE)
         {
          in2 = TRUE;
          strcpy(temp, temp_value);
         }
        else
         {
          subnon_ptr1 = subnon_ptr1->snn_next_node;
          subnon_ptr2 = subnon_ptr1;
         }
        }
      while ((dernon_ptr != NULL) && (in1 == FALSE) && (in2 == FALSE) &&
(in3 == FALSE))
        {
        if(strcmp(dernon_ptr2->dnn_name, temp_value) == FALSE)
         {
          in3 = TRUE;
          strcpy(temp,temp_value);
         }
        else
         {
          dernon_ptr1 = dernon_ptr1->dnn_next_node;
          dernon_ptr2 = dernon_ptr1;
         }
        }
      }
    derived_range
     {
      check_ids = TRUE;
     }
```

```
                ;

derived_range: integer_type_definition

          /* the type is now checked to see if in fact the type field */
          /* in the kms_info structure actually contains the value    */
          /* identified in derived_type_definition above              */

            {
             if (in1 == TRUE)
              {
               if(strcmp(kms_ptr->dki_ent_non.enn_name, temp) != FALSE)
                 proc_eval_error(serror);
              }
             else
              {
               if (in2 == TRUE)
                {
                 if(strcmp(kms_ptr->dki_sub_non.snn_name, temp) != FALSE)
                   proc_eval_error(serror);
                }
               else
                {
                 if (in3 == TRUE)
                  {
                   if(strcmp(kms_ptr->dki_der_non.dnn_name, temp) != FALSE)
                    proc_eval_error(serror);
                  }
                 else
                  {
                   proc_eval_error(serror);
                  }
                }
              }
            }
          | real_type_definition
            {
             if (in1 == TRUE)
              {
               if(strcmp(kms_ptr->dki_ent_non.enn_name, temp) != FALSE)
```

```
        proc_eval_error(serror);
      }
     else
      {
       if (in2 == TRUE)
        {
         if(strcmp(kms_ptr->dki_sub_non.snn_name, temp) != FALSE)
           proc_eval_error(serror);
        }
       else
        {
         if (in3 == TRUE)
          {
           if(strcmp(kms_ptr->dki_der_non.dnn_name, temp) != FALSE)
            proc_eval_error(serror);
          }
         else
          {
           proc_eval_error(serror);
          }
        }
      }
    }
    ;

entity_type_definition: EMPTY
                      | ENTITY
                         {
                          check_ids = TRUE;
                         }
                        entity_component_declaration_list
                         {
                          check_ids = FALSE;
                         }
                        end_entity
                        ;

entity_component_declaration_list: entity_component_declaration
                                 | entity_component_declatation_list entity_component_declatation
```

```
                ;

end_entity: END
        | END ENTITY
        ;

entity_component_declatation:
                        {
                         serror = 5;
                        }
  *************              identifier_list COLON
                        {
                         check_ids = FALSE;
                        }
                       function_type default_value SEMICOLON
                        ;

function_type: common_type
          | FLOAT
           {
            kms_ptr->dki_funct.fn_type = 'f';
            kms_ptr->dki_funct.fn_range = FALSE;
            kms_ptr->dki_funct.fn_total_length = FLTlength;
            kms_ptr->dki_funct.fn_num_value = 0;
            kms_ptr->dki_funct.fn_value = NULL;
            kms_ptr->dki_funct.fn_entptr = NULL;
            kms_ptr->dki_funct.fn_subptr = NULL;
            kms_ptr->dki_funct.fn_nonentptr = NULL;
            kms_ptr->dki_funct.fn_nonsubptr = NULL;
            kms_ptr->dki_funct.fn_nonderptr = NULL;
            kms_ptr->dki_funct.fn_next_fntptr = NULL;
            kms_ptr->dki_funct.fn_entnull = FALSE;
            kms_ptr->dki_funct.fn_unique = FALSE;
           }
          | INTEGER
           {
            kms_ptr->dki_funct.fn_type = 'i';
            kms_ptr->dki_funct.fn_range = FALSE;
            kms_ptr->dki_funct.fn_total_length = INTlength;
```

```
        kms_ptr->dki_funct.fn_num_value = 0;
        kms_ptr->dki_funct.fn_value = NULL;
        kms_ptr->dki_funct.fn_entptr = NULL;
        kms_ptr->dki_funct.fn_subptr = NULL;
        kms_ptr->dki_funct.fn_nonentptr = NULL;
        kms_ptr->dki_funct.fn_nonsubptr = NULL;
        kms_ptr->dki_funct.fn_nonderptr = NULL;
        kms_ptr->dki_funct.fn_next_fntptr = NULL;
        kms_ptr->dki_funct.fn_entnull = FALSE;
        kms_ptr->dki_funct.fn_unique = FALSE;
       }
      | BOOLEAN
       {
        kms_ptr->dki_funct.fn_type = 'b';
        kms_ptr->dki_funct.fn_range = FALSE;
        kms_ptr->dki_funct.fn_total_length = BOOLlength;
        kms_ptr->dki_funct.fn_num_value = 0;
        kms_ptr->dki_funct.fn_value = NULL;
        kms_ptr->dki_funct.fn_entptr = NULL;
        kms_ptr->dki_funct.fn_subptr = NULL;
        kms_ptr->dki_funct.fn_nonentptr = NULL;
        kms_ptr->dki_funct.fn_nonsubptr = NULL;
        kms_ptr->dki_funct.fn_nonderptr = NULL;
        kms_ptr->dki_funct.fn_next_fntptr = NULL;
        kms_ptr->dki_funct.fn_entnull = FALSE;
        kms_ptr->dki_funct.fn_unique = FALSE;
       }
      | set_type_definition
      ;

common_type: enumeration_type_definition
     | integer_type_definition
     | real_type_definition
     | name_id
       {
        non_ent_ptr1 = db_ptr->edn_nonentity;
        non_ent_ptr2 = non_ent_ptr1;
        ent_ptr1 = db_ptr->edn_entity;
```

```
ent_ptr2 = ent_ptr1;
gen_ptr1 = db_ptr->edn_subptr;
gen_ptr2 = gen_ptr1;
subnon_ptr1 = db_ptr->edn_nonsubptr;
subnon_ptr2 = subnon_ptr1;
dernon_ptr1 = db_ptr->edn_nonderptr;
dernon_ptr2 = dernon_ptr1;
in1 = FALSE;
in2 = FALSE;
in3 = FALSE;
in4 = FALSE;
in5 = FALSE;
while ((non_ent_ptr2 != NULL) && (in1 == FALSE))
  {
   if(strcmp(non_ent_ptr2->enn_name, temp_value) == FALSE)
    {
     in1 = TRUE;
     strcpy(temp, temp_value);
    }
   else
    {
     non_ent_ptr1 = non_ent_ptr1->enn_next_node;
     non_ent_ptr2 = non_ent_ptr1;
    }
  }
while ((subnon_ptr2 != NULL) && (in1 == FALSE) && (in2 == FALSE))
  {
   if(strcmp(subnon_ptr2->snn_name, temp_value) == FALSE)
    {
     in2 = TRUE;
     strcpy(temp, temp_value);
    }
   else
    {
     subnon_ptr1 = subnon_ptr1->snn_next_node;
     subnon_ptr2 = subnon_ptr1;
    }
  }
```

```
while ((dernon_ptr2 != NULL) && (in1 == FALSE) && (in2 == FALSE) && (in3
== FALSE))
 {
  if(strcmp(dernon_ptr2->dnn_name, temp_value) == FALSE)
   {
    in3 = TRUE;
    strcpy(temp,temp_value);
   }
  else
   {
    dernon_ptr1 = dernon_ptr1->dnn_next_node;
    dernon_ptr2 = dernon_ptr1;
   }
 }
while((gen_ptr2 != NULL) && (in1 == FALSE) && (in2 == FALSE) && (in3 ==
FALSE) && (in4 == FALSE))
 {
  if(strcmp(gen_ptr2->gsn_name, temp_value) == FALSE)
   {
    in4 = TRUE;
    strcpy(temp, temp_value);
   }
  else
   {
    gen_ptr1 = gen_ptr1->gsn_next_genptr;
    gen_ptr2 = gen_ptr1;
   }
 }
while((ent_ptr2 != NULL) && (in1 == FALSE) && (in2 == FALSE) && (in3 ==
FALSE) && (in4 == FALSE) && (in5 == FALSE))
 {
  if(strcmp(ent_ptr2->en_name, temp_value) == FALSE)
   {
    in5 = TRUE;
    strcpy(temp, temp_value);
   }
  else
   {
```

```
      ent_ptr1 = ent_ptr1->en_next_ent;
      ent_ptr2 = ent_ptr1;
     }
   }
 if (in1 == TRUE)
  {
   func_ptr1->fn_nonentptr = non_ent_ptr2;
   kms_ptr->dki_funct.fn_type = 'v';
  }
 else
  {
   if (in2 == TRUE)
    {
     func_ptr1->fn_nonsubptr = subnon_ptr2;
     kms_ptr->dki_funct.fn_type = 'w';
    }
   else
    {
     if (in3 == TRUE)
      {
       func_ptr1->fn_nonderptr = dernon_ptr2;
       kms_ptr->dki_funct.fn_type = 'x';
      }
     else
      {
       if (in4 == TRUE)
        {
         func_ptr1->fn_subptr = gen_ptr2;
         kms_ptr->dki_funct.fn_type = 'y';
        }
       else
        {
         if(in5 == TRUE)
          {
           func_ptr1->fn_entptr = ent_ptr2;
           kms_ptr->dki_funct.fn_type = 'z';
          }
         else
```

```
                {
                 proc_eval_error(serror);
                }
               }
              }
             }
            }
           }
          | string_rule
          ;

string_rule: STRING LP
           {
            kms_ptr->dki_funct.fn_type = 's';
            kms_ptr->dki_funct.fn_range = TRUE;
            kms_ptr->dki_funct.fn_num_value = 2;
           }
          int_range RP
          ;

set_type_definition: SET OF commmon_type
                {    ***********         kms_ptr->dki_funct.fn_type    =    toupper(kms_ptr-
>dki_funct.fn_type);
                }
                ;

default_value: empty
           | ASSIGN simple_const SEMICOLON
           ; ***********

identifier_list: name_id
              {
               temp_ptr = kms_ptr->dki_temp_ptr;
               free_ident_list(temp_ptr);
               kms_ptr->dki_temp_ptr = dap_ident_list_alloc();
               temp_ptr = kms_ptr->dki_temp_ptr;
               strcpy(temp_ptr->il_name, temp_value);
               temp_ptr->il_next = NULL;
              }
             | identifier_list COMMA name_id
```

```
        {
         new_temp_ptr = temp_ptr;
         while (new_temp_ptr->il_next != NULL)
           new_temp_ptr = new_temp_ptr->il_next;
         new_temp_ptr->il_next = dap_ident_list_alloc();
         new_temp_ptr = new_temp_ptr->il_next;
         strcpy(new_temp_ptr->il_name, temp_value);
         new_temp_ptr->il_next = NULL;
        }
       ;


subtype_declaration: SUBTYPE
            {
             curr_op = SubTypeIs;
             check_ids = TRUE;
             serror  = 7;
            }
           name_id
            {
             serror = 8;
             check_ids = FALSE;
            }
           IS subtype_indication SEMICOLON
        | SUBTYPE
            {
             check_ids = FALSE;
             curr_op = SubTypeIs;
             serror = 10;
            }
          name_id
            {
             gen_ptr1 = db_ptr->edn_subptr;
             gen_ptr2 = gen_ptr1;
             in1 = FALSE;
             while ((gen_ptr2 != NULL) && (in1 == FALSE))
               {
                if (strcmp(gen_ptr2->gsn_name, temp_value) == FALSE)
                  in1 = TRUE;
```

```
        else
         {
          gen_ptr1 = gen_ptr1->gsn_next_genptr;
          gen_ptr2 = gen_ptr1;
         }
        }
      if (in1 == FALSE)
        proc_eval_error(serror);
     }
   IS name1_list
     {
      in1 = FALSE;
      in2 = FALSE;
      in3 = FALSE;
      name1_ptr1 = kms_ptr->dki_name1_ptr;
      name1_ptr2 = name1_ptr1;
      gen_ptr1 = db_ptr->edn_subptr;
      gen_ptr2 = gen_ptr1;
      overlapsub_ptr1 = gen_ptr1->gsn_subptr;
      overlapsub_ptr2 = overlapsub_ptr1;
      while (name1_ptr2 != NULL)
        {
         while ((gen_ptr2 != NULL) && (in1 == FALSE))
           {
            if (strcmp(gen_ptr2->gsn_name, name1_ptr2->il_name) == FALSE)
             {
              in1 = TRUE;
              in2 = TRUE;
              if (overlapsub_ptr2 == NULL)
               {
                gen_ptr1 = gen_ptr1->gsn_subptr;
                gen_ptr2 = gen_ptr1;
                gen_ptr2 = dap_overlap_sub_node_alloc();
                overlapsub_ptr1 = gen_ptr2;
                overlapsub_ptr2 = overlapsub_ptr1;
                name1_ptr2->il_name = overlapsub_ptr2;
                overlapsub_ptr2->osn_name = name1_ptr2->il_name;
                overlapsub_ptr2->oen_next_name = NULL;
```

```
      }
     else
      {
       while (overlapsub_ptr2->oen_name != NULL)
         overlapsub_ptr2 = overlapsub_ptr2->oen_next_name;
       overlapsub_ptr2->oen_next_name = dap_overlap_sub_node_alloc();
       overlapsub_ptr1 = overlapsub_ptr2->oen_next_name;
       overlapsub_ptr2 = overlapsub_ptr1;
       name1_ptr2->il_name = overlapsub_ptr2;
       overlapsub_ptr2->osn_name = name1_ptr2->il_name;
       overlapsub_ptr2->oen_next_name = NULL;
      }
    }
   else
    {
     gen_ptr1 = gen_ptr1->gsn_next_genptr;
     gen_ptr2 = gen_ptr1;
    }
  }
ent_ptr1 = db_ptr->edn_entity;
ent_ptr2 = ent_ptr1;
overlapent_ptr1 = gen_ptr1->gsn_entptr;
overlapent_ptr2 = overlapent_ptr1;
while ((ent_ptr2 != NULL) && (in3 == FALSE))
  {
   if (strcmp(ent_ptr2->gsn_name, name1_ptr2->il_name) == FALSE)
    {
     in3 = TRUE;
     in4 = TRUE;
     if (overlapsub_ptr2 == NULL)
      {
       gen_ptr1 = gen_ptr1->gsn_entptr;
       gen_ptr2 = gen_ptr1;
       gen_ptr2 = dap_overlap_ent_node_alloc();
       overlapent_ptr1 = gen_ptr2;
       overlapent_ptr2 = overlapent_ptr1;
       name1_ptr2->il_name = overlapent_ptr2;
       overlapent_ptr2->oen_name = name1_ptr2->il_name;
```

```
            overlapent_ptr2->oen_next_name = NULL;
           }
          else
           {
            while (overlapent_ptr2->oen_name != NULL)
              overlapent_ptr2 = overlapent_ptr2->oen_next_name;
            overlapent_ptr2->oen_next_name = dap_overlap_ent_node_alloc();
            overlapent_ptr1 = overlapent_ptr2->oen_next_name;
            overlapent_ptr2 = overlapent_ptr1;
            name1_ptr2->il_name = overlapent_ptr2;
            overlapent_ptr2->oen_name = name1_ptr2->il_name;
            overlapent_ptr2->oen_next_name = NULL;
           }
         }
        else
         {
          ent_ptr1 = ent_ptr1->en_next_ent;
          ent_ptr2 = ent_ptr1;
         }
       }
      if ((in2 == FALSE) && (in4 == FALSE))
        proc_eval_error(serror);
      else
       {
        in1 = FALSE;
        in2 = FALSE;
        in3 = FALSE;
        in4 = FALSE;
        gen_ptr1 = db_ptr->edn_subptr;
        gen_ptr2 = gen_ptr1;
        name1_ptr1 = name1_ptr1->il_next;
        name1_ptr2 = name1_ptr1;
       }
     }
    check_ids = TRUE;
   }
********   entity_type_definition SEMICOLON

         | incomplete_subtype_declaration
```

```
            ;

subtype_indication: name_id subtype_definition
                ;

subtype_definition: RANGE enumeration_literal ELIPSES enumeration_literal
                | integer_type_definition
                | real_type_definition
                | empty
                  {
                   node_type = find_previous(temp_value, non_ent_ptr1, subnon_ptr1, dernon_ptr1);

                   switch(node_type)
                    {
                     case NonEnt:
                          subnon_ptr1 = dap_sub_non_node_alloc();
                          strcpy(subnon_ptr1->snn_name, ent_non_ptr1->enn_name);
                          subnon_ptr1->snn_type = ent_non_ptr1->enn_type;
                          subnon_ptr1->snn_total_length = ent_non_ptr1->enn_total_length;
                          subnon_ptr1->snn_range = ent_non_ptr1->enn_range;
                          subnon-ptr1->snn_num_values = ent_non_ptr1->enn_num_values;
                          subnon_ptr1->snn_value = ent_non_ptr1->enn_value;
                          subnon_ptr1->snn_next_node = ent_non_ptr1->enn_next_node;
                          subnon_ptr1 = db_ptr->edn_nonsubptr;
                          subnon_ptr2 = subnon_ptr1;
                          if (subnon_ptr2 == NULL)
                            db_ptr->edn_nonsubptr = subnon_ptr1;
                          else
                           {
                            while (subnon_ptr2->snn_next_node != NULL)
                              subnon_ptr2 = subnon_ptr2->snn_next_node;
                            subnon_ptr2->snn_next_node = subnon_ptr1;
                           }
                          subnon_ptr1 = NULL;
                          break;

                     case Derived:
                          subnon_ptr1 = dap_sub_non_node_alloc();
                          strcpy(subnon_ptr1->snn_name, dernon_ptr1->dnn_name);
                          subnon_ptr1->snn_type = dernon_ptr1->dnn_type;
```

```
      subnon_ptr1->snn_total_length = dernon_ptr1->dnn_total_length;
      subnon_ptr1->snn_range = dernon_ptr1->dnn_range;
      subnon-ptr1->snn_num_values = dernon_ptr1->dnn_num_values;
      subnon_ptr1->snn_value = dernon_ptr1->dnn_value;
      subnon_ptr1->snn_next_node = dernon_ptr1->dnn_next_node;
      subnon_ptr1 = db_ptr->edn_nonsubptr;
      subnon_ptr2 = subnon_ptr1;
      if (subnon_ptr2 == NULL)
        db_ptr->edn_nonsubptr = subnon_ptr1;
      else
       {
        while (subnon_ptr2->snn_next_node != NULL)
          subnon_ptr2 = subnon_ptr2->snn_next_node;
        subnon_ptr2->snn_next_node = subnon_ptr1;
       }
      subnon_ptr1 = NULL;
      break;

  case SubNon:
      subnon_ptr1 = dap_sub_non_node_alloc();
      strcpy(subnon_ptr1->snn_name, subnon_ptr1->snn_name);
      subnon_ptr1->snn_type = subnon_ptr1->snn_type;
      subnon_ptr1->snn_total_length = subnon_ptr1->snn_total_length;
      subnon_ptr1->snn_range = subnon_ptr1->snn_range;
      subnon-ptr1->snn_num_values = subnon_ptr1->snn_num_values;
      subnon_ptr1->snn_value = subnon_ptr1->snn_value;
      subnon_ptr1->snn_next_node = subnon_ptr1->snn_next_node;
      subnon_ptr1 = db_ptr->edn_nonsubptr;
      subnon_ptr2 = subnon_ptr1;
      if (subnon_ptr2 == NULL)
        db_ptr->edn_nonsubptr = subnon_ptr1;
      else
       {
        while (subnon_ptr2->snn_next_node != NULL)
          subnon_ptr2 = subnon_ptr2->snn_next_node;
        subnon_ptr2->snn_next_node = subnon_ptr1;
       }
      subnon_ptr1 = NULL;
      break;
```

```
                }
              }
              ;

type_mark: name1
        | predefined_tm
        ;

name1_list: name1
              {
               temp_ptr = kms_ptr->dki_temp_ptr;
               free_ident_list(temp_ptr);
               kms_ptr->dki_temp_ptr = dap_ident_list_alloc();
               temp_ptr = kms_ptr->dki_temp_ptr;
               strcpy(temp_ptr->il_name, temp_value);
               temp_ptr->il_next = NULL;
              }
          | name1_list COMMA name1
              {
               new_temp_ptr = temp_ptr;
               while (new_temp_ptr->il_next != NULL)
                 new_temp_ptr = new_temp_ptr->il_next;
               new_temp_ptr->il_next = dap_ident_list_alloc();
               new_temp_ptr = new_temp_ptr->il_next;
               strcpy(new_temp_ptr->il_name, temp_value);
               new_temp_ptr->il_next = NULL;
              }
          ;

name1: name_id
     | selected_component
     ;

predefined_tm: STRING     /* for type values in declaration */
           | INTEGER
           | BOOLEAN
           | FLOAT
           ;
```

```
ada_range: simple_expression ELIPSES simple_expression
        ;


incomplete_subtype_declaration: SUBTYPE
                        {
                         check_ids = TRUE;
                        }
                        /* entity */
                        name_id SEMICOLON
                        {
                         in = FALSE;
                         gen_ptr1 = db_ptr->edn_subptr;
                         while (gen_ptr1 != NULL)
                           {
                            if (strcmp(gen_ptr1->gsn_name, temp_value) == FALSE)
                             {
                              proc_eval_error(serror);
                              in = TRUE;
                             }
                            else
                              gen_ptr1 = gen_ptr1->gsn_next_genptr;
                              gen_ptr2 = gen_ptr1;
                           }

                         if (in == FALSE)
                          {
                           gen_ptr2 = dap_gen_node_alloc();
                           strcpy(gen_ptr2->gsn_name, temp_value);
                           gen_ptr2->gsn_num_funct = 0;
                           gen_ptr2->gsn_terminal = FALSE;
                           gen_ptr2->gsn_entptr = NULL;
                           gen_ptr2->gsn_num_ent = NULL;
                           gen_ptr2->gsn_ftnptr = NULL;
                           gen_ptr2->gsn_subptr = NULL;
                           gen_ptr2->gsn_num_sub = 0;
                           gen_ptr2->gsn_next_genptr;

                           gen_ptr1 = db_ptr->edn_subptr;
                           gen_ptr2 = gen_ptr1;
```

```
                    if (gen_ptr2 == NULL)
                     {
                       db_ptr->edn_subptr = gen_ptr1;
                       gen_ptr2 = db_ptr->edn_subptr;
                       gen_ptr2 = NULL;
                     }
                    else
                     {
                       gen_ptr1 = db_ptr->edn_subptr;
                       gen_ptr2 = gen_ptr1;
                       while (gen_ptr2->gsn_next_genptr != NULL)
                         gen_ptr2 = gen_ptr2->gsn_next_genptr;
                       gen_ptr1->gsn_next_genptr = gen_ptr2;
                       gen_ptr2 = NULL;
                     }
                   }
                  check_ids = FALSE;
                 }
                ;

null_value_constraint: WITHNULL
               | WITHOUTNULL
               ;

selected_component: IDENTIFIER  DOT IDENTIFIER
              ;

attribute: type_mark HYPHEN LP loop_parameter RP
        {
        }
       | type_mark HYPHEN attribute_identifier LP ada_expresssion RP
       ;

attribute_identifier: IMAGE
               | VAL
               | POS
               | VALUE
               ;

literal: NUMERIC_LITERAL    /* for user default type values in declaration */
```

```
     | LITERAL_CHARACTER
     | CHARACTER_STRING
     | NULL
     | TRUE
     | FALSE
     ;

named_aggregate: LP
            {
            }
            component_association_list RP
          | EMPTY    /* to handle case when no attributes listed in */
                  /* CREATE due to LR(1) grammar */
          ;

component_association_list: component_association
                  {
              | component_association_list COMMA component_association
                  ;

component_association: identifier_choice
               {
                 CREATE:
               }
               IMPLY ada_expresssion
              ;

identifier_choice: IDENTIFIER
            {
            }
           | identifier_choice  IDENTIFIER
           ;

ada_expression: relation
          | rel_or_list
          | rel_xor_list
          | rel_and_then_list
          | rel_or_else_list
          ;

rel_and_list: relation AND relation
```

```
          | rel_and_list AND relation
          ;

rel_or_list: relation OR relation
          | rel_or_list OR relation
          ;

rel_xor_list: relation XOR relation
          | rel_xor_list XOR relation
          ;

rel_and_then_list: relation AND THEN relation
               | rel_and_then_list AND THEN relation
               ;

rel_or_else_list: relation OR ELSE relation
              | rel_or_else_list OR ELSE relation
              ; relation: simple_expression
        {
          CREATE:
            (FOR | FOR EACH) & DESTROY
         relational_operator
            CREATE:
            (FOR | FOR EACH) & DESTROY:
         simple_expression
        | expr_in_op ada_range
        | expr_in_type_mark
        | simple_expression test_set
        | quantification_clause_list simple_expression
        ;

simple_expression: term_list
               | unary_operator term_list /* probably won't use since */
               | set_exp_list            /* involves expressions    */
               ;

set_exp_list: primary set_operator primary
          | set_exp_list set_operator primary
          ;

primary: ada_name2
      | primary2
```

```
        ;

primary2: literal
        {
          /* NUMERIC_LITERAL | LITERAL_CHARACTER | CHARACTER_STRING | */
          /* NULL | TRUE | FALSE */
        }
       | set_constructor
       | LP ada_expression RP
       | indexed_component
       ;

  /*  &c type_conversion is handled by indexed_component */

indexed_component: ada_name
                ;

ada_name: ada_name2
       | indexed_component
       ;

ada_name2: type_mark
         /* -> name1|predefined_tm(b,s,i,f) */
        | function_call
        ;

term_list: term
        | term_list adding_operator term
        ;

term: factor_list
   ;

factor_list: factor
          | factor_list multiplying_operator factor
          ;

factor: primary
      | primary EXPONENT primary
      ;

quantification_clause_list: quantification_clause COLON
                        | quantification_clause_list
```

```
                    quantification_clause COLON
                    ;

quantification_clause: FOR quantifier IDENTIFIER IN domain
                 ;

quantifier: SOME
     | EVERY
     | NO
     ;

domain: primary
   | primary WHERE
  | and ((simple_exp1_list 2nd_on_simple_exp1_list simple_exp2_list) |
     | and ((simple_exp1_list 2nd_on_simple_exp1_list simple_exp2_list) |
   ;

expr_in_op: simple_expression
       CREATE:
      in_op
     ;

expr_in_type_mark: expr_in_op type_mark
         ;

test_set: isin_operator primary
    | is_op EMPTY
    ;

relational_operator: =
         | /=
         | &<
         | &<=
         | &>
         | &>=
         | EQ
         | NE
         | NQ
         | LT
         | LE
         | LQ
```

```
                | GT
                | GE
                | GQ
                ;

adding_operator: +
              | -
              | &&  /* concat */
              ;

unary_operator: arith_unary_op
             | log_unary_op
             ;

arith_unary_op: +
             | -
             ;

log_unary_op: NOT
            ;

multiplying_operator: *
                   | /
                   | MOD
                   | REM     /* remainder */
                   ;

in_op: IN
     | NOT IN
     ;

is_op: IS
     | IS NOT
     ;

isin_operator: is_op IN
             ;

set_operator: UNION
            | diff_op
            | inter_op
            ;
```

```
diff_op: DIFF | DIFFERENCE
    ;

inter_op: INTER
    | INTERSECT
    | INTERSECTION
    ;

set_constructor:
        LCB
         RCB
        | LCB RCB      /* empty or null list */
        | LCB
        | LCB expr_in_op primary2 WHERE condition RCB
              /* because of primary2, probably not for CREATE */
        ;

function_call: predefined_function_call
        ;

predefined_function_call: function_name
              | attribute
              ;

aggregate_argument: primary
          | IDENTIFIER DUPLICATES LP primary RP
          ;

function_name: COUNT
        | SUM
        | AVG
        | MIN
        | MAX
        ; **************************************************************
   End of ddl_statement ****************************************************************

dml_statement: simple_statement
        | compound_statement
        ;

simple_statement: exit_statement
```

```
            | assignment_statement
            | create_statement
            | include_statement
            | exclude_statement
            | destroy_statement
            | move_statement
            | procedure_call
            ;

exit_statement: EXIT end_exit SEMICOLON
           ;

end_exit: IDENTIFIER
      | WHEN condition
      | IDENTIFIER WHEN condition
      ;

assignment_statement: indexed_component ASSIGN ada_expression
                ;

create_statement: CREATE NEW
             {
              check_ids = FALSE;
             }
            name1_list
             {
              temp_ptr = kms_ptr->dki_temp_ptr;
              while (temp_ptr != NULL)
                {
                 in1 = FALSE;
                 ent_ptr1 = db_ptr->edn_entity;
                 while ((ent_ptr1 != NULL) && (in1 == FALSE))
                   {
                    if (strcmp(temp_ptr->il_name, ent_ptr1->en_name) == FALSE)
                     {
                      create_list1 = dap_create_list_alloc(); must create
                      create_list1->dcl_node_type = Entity;
                      create_list1->dcl_name = ent_ptr1->en_name;
```

```
        create_list1->dcl_ent_ptr = ent_ptr1;
        create_list1->dcl_sub_ptr = NULL;
        create_list1->dcl_next = NULL;
        create_list2 = kms_ptr->dki_create.dci_create;
        if (create_list2 == NULL)
         {
          kms_ptr->dki_create.dci_create = create_list1;
          create_list2 = NULL;
         }
        else
         {
          while (create_list2->dcl_next != NULL)
            create_list2 = create_list2->dcl_next;
          create_list2->dcl_next = create_list1;
          create_list2 = NULL;
         }
        in1 = TRUE;
       }
      else
       {
        ent_ptr1 = ent_ptr1->en_next_ent;
       }
     }
    temp_ptr = temp_ptr->il_next;
  }
 temp_ptr = kms_ptr->dki_temp_ptr;
 while (temp_ptr != NULL)
  {
   in2 = FALSE;
   gen_ptr1 = db_ptr->edn_subptr;
   while ((gen_ptr1 != NULL) && (in2 == FALSE))
     {
      if (strcmp(temp_ptr->il_name, gen_ptr1->gsn_name) == FALSE)
       {
        create_list1 = dap_create_list_alloc(); must create
        create_list1->dcl_node_type = GenSub;
        create_list1->dcl_name = gen_ptr1->gsn_name;
        create_list1->dcl_ent_ptr = NULL;
```

```
            create_list1->dcl_sub_ptr = gen_ptr1;
            create_list1->dcl_next = NULL;
            create_list2 = kms_ptr->dki_create.dci_create;
            if (create_list2 == NULL)
             {
              kms_ptr->dki_create.dci_create = create_list1;
              create_list2 = NULL;
             }
            else
             {
              while (create_list2->dcl_next != NULL)
                create_list2 = create_list2->dcl_next;
              create_list2->dcl_next = create_list1;
              create_list2 = NULL;
             }
            ptr = gen_ptr1;
            proc_create(ptr);
            in2 = TRUE;
           }
          else
           {
            gen_ptr1 = gen_ptr1->gsn_next_genptr;
           }
          }
        temp_ptr = temp_ptr->il_next;
       }

/* the recursive procedure follows */

      proc_create(ptr);

      {
       gen_ptr2 = ptr;
       if (gen_ptr2->gsn_entptr != NULL)
        {
         overlapent_ptr1 = gsn_entptr;
         ent_ptr2 = overlapent_ptr1->oen_name;
         while (overlapent_ptr1 != NULL)
          {
           create_list1 = dap_create_list_alloc();  must create
```

```
        create_list1->dcl_node_type = Entity;
        create_list1->dcl_name = ent_ptr2->en_name;
        create_list1->dcl_ent_ptr = ent_ptr2;
        create_list1->dcl_sub_ptr = NULL;
        create_list1->dcl_next = NULL;
        create_list2 = kms_ptr->dki_create.dci_create;
        if (create_list2 == NULL)
         {
          kms_ptr->dki_create.dci_create = create_list1;
          create_list2 = NULL;
         }
        else
         {
          while (create_list2->dcl_next != NULL)
            create_list2 = create_list2->dcl_next;
          create_list2->dcl_next = create_list1;
          create_list2 = NULL;
         }
        overlapent_ptr1 = overlapent_ptr1->oen_next_name;
       }
     }
   if (gen_ptr2->gsn_subptr != NULL)
    {
     overlapsub_ptr1 = gsn_subptr;
     gen_ptr1 = overlapsub_ptr1->osn_name;
     while (overlapsub_ptr1 != NULL)
       {
        create_list1 = dap_create_list_alloc();  must create
        create_list1->dcl_node_type = GenSub;
        create_list1->dcl_name = gen_ptr1->gsn_name;
        create_list1->dcl_ent_ptr = NULL;
        create_list1->dcl_sub_ptr = gen_ptr1;
        create_list1->dcl_next = NULL;
        create_list2 = kms_ptr->dki_create.dci_create;
        if (create_list2 == NULL)
         {
          kms_ptr->dki_create.dci_create = create_list1;
          create_list2 = NULL;
```

```
            }
            else
            {
              while (create_list2->dcl_next != NULL)
                create_list2 = create_list2->dcl_next;
              create_list2->dcl_next = create_list1;
              create_list2 = NULL;
            }
           proc_create(overlapsub_ptr1->osn_next_name);
           overlapsub_ptr1 = overlapsub_ptr1->osn_next_name;
          }
        }
 /*  end recursive procedure  */

        }
        named_aggregate SEMICOLON
        {
        }
        ;

allocator: NEW
      name1_list    /* rule modified to LL(1) from LR(1) */
      named_aggregate
    ;

include_statement: INCLUDE ada_expression
          INTO indexed_component
          SEMICOLON
        ;

exclude_statement: EXCLUDE ada_expression
          FROM indexed_component
          SEMICOLON
        ;

destroy_statement: DESTROY
          ada_expression SEMICOLON
        ;
```

```
move_statement: MOVE ada_expression move_from SEMICOLON
           | MOVE ada_expression move_to SEMICOLON
           | MOVE ada_expression move_from move_to SEMICOLON
           ;

move_from: FROM name1_list
       ;

move_to: INTO name1_list
      | INTO name1_list named_aggregate
      ;

procedure_call: procedure_name SEMICOLON
           | procedure_name parameter_part SEMICOLON
           ;

parameter_part: LP ada_expression_list RP
           ;

procedure_name: PRINT
           | PRINT_LINE
           | CANCEL
           | HEADER_PRINT_LINE
           | FORMAT
           | FORMAT_LINE
           | HEADER_FORMAT_LINE
           ;

compound_statement: if_statement
               | atomic_statement
               | loop_statement
               ;

if_statement: if_part end_if SEMICOLON
          | if_part elsif_list end_if SEMICOLON
          | if_part else_part end_if SEMICOLON
          | if_part elsif_list else_part end_if SEMICOLON
          ;
```

```
if_part: IF condition THEN sequence_of_statements
      ;

elsif_list: elsif_part
        | elsif_list   elsif_part
        ;

elsif_part: ELSIF condition THEN sequence_of_statements
        ;

else_part: ELSE sequence_of_statements
       ;

end_if: END
     | END IF
     ;

atomic_statement: begin_atomic sequence_of_statements
             end_atomic SEMICOLON
           ;

begin_atomic: ATOMIC
         | IDENTIFIER  COLON  ATOMIC
         ;

end_atomic: END
        | END ATOMIC
        | END IDENTIFIER
        | END ATOMIC IDENTIFIER
        ;

loop_statement:
           real_loop SEMICOLON
          | IDENTIFIER COLON real_loop IDENTIFIER SEMICOLON
          | IDENTIFIER COLON real_loop SEMICOLON
          | real_loop IDENTIFIER
          ;

real_loop: iteration_clause basic_loop end_loop
```

```
basic_loop: seqence_of_statements
        | LOOP sequence_of_statements
        ;

end_loop: END
       | END LOOP
       ;

iteration_clause: iteration_body
                | iteration_body order_by_clause
                ;

iteration_body: for_clause loop_parameter IN domain
              ;

for_clause: FOR
          | FOR EACH
          ;

loop_parameter: IDENTIFIER

order_by_clause: BY order_component_list
               ;

order_component_list: order_component
                    | order_component_list COMMA order_component
                    ;

order_component: indexed_component
               | sort_order indexed_component
               ;

sort_order: ASCENDING
          | DESCENDING
          ;

sequence_of_statements: dml_statement
                      | sequence_of_statements dml_statement
                      ;
```

```
condition: ada_expression
        ;

generalized_expression_list: generalized_expression
                       | generalized_expression_list COMMA
                         generalized_expression
                       ;

generalized_expression: ada_expression
                   | ada_range
                   ;

ada_expression_list: ada_expression
                | ada_expression_list COMMA ada_expression
                ;
```

LIST OF REFERENCES

1. Demurjian, S. A. and Hsiao, D. K., "New Directions in Database-Systems Research and Development," in the *Proceedings of the New Directions in Computing Conference*, Trondheim, Norway, pp. 188-197, August, 1985; also in Technical Report, NPS52-85-001, Naval Postgraduate School, Monterey, California, February 1985.

2. Hsiao, D. K., and Harary, F., "A Formal System for Information Retrieval from Files," *Communications of the ACM,* Vol. 13, No. 2, pp. 67-73, February 1970, also in Corrigenda, Vol 13., No. 4, April 1970.

3. Wong, E., and Chiang, T. C., "Canonical Structure in Attribute Based File Organization," *Communications of the ACM,* pp. 593-597, September 1971.

4. Rothnie, J. B. Jr., "Attribute Based File Organization in a Paged Memory Environment," *Communications of the ACM,* Vol. 17, No. 2, pp. 63-69, February 1974.

5. The Ohio State University, Columbus, Ohio, Technical Report No. OSU-CISRC-TR-77-7, *DBC Software Requirements for Supporting Relational Databases,* by J. Banerjee and D. K. Hsiao, November 1977.

6. Naval Postgraduate School, Monterey, California, Technical Report, NPS52-85-002, *A Multi-Backend Database System for Performance Gains, Capacity Growth and Hardware Gains,* by S. A. Demurjian, D. K. Hsiao and J. Menon, February 1985.

7. Fox, S., Landers, T., Ries, D.R., and Rosenberg, R.L., *Daplex User's Manual, CCA-84-01,* Computer Corporation of America, June 1984.

8. Date, C.J., *An Introduction to Database Systems,* 3d ed., Addison Wesley, pp. 117-142, 1982.

9. Boehm, B.W., *Software Engineering Economics,* Prentice-Hall, pp. 14-46, 1981.

10. Naval Postgraduate School, Monterey, California, Technical Report, NPS52-84-012, *Software Engineering Techniques for Large-Scale Database Systems as Applied to the Implementation of a Multi-Backend Database System,* by Ali Orooji, Douglas Kerr and David K. Hsiao, pp. 27, August 1984.

11. Goisman, P.L., *The Design and Analysis of a Complete Entity-Relationship Interface for the Multi-Lingual Database System,* M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1985.

12. Benson, T.P., and Wentz, G.L., *The Design and Implementation of a Hierarchical Interface for the Multi-Lingual Database System,* M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1985.

13. Kloepping, G.R., and Mack, J.F., *The Design and Implementation of a Relational Interface for the Multi-Lingual Database System,* M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1985.

14. Kernighan, B.W., and Ritchie, D.M., *The C Programming Language,* Prentice-Hall, 1978.

15. Howden, W.E., "Reliability of the Path Analysis and Testing Strategy," *IEEE Transactions on Software Engineering,* Vol. SE-2, pp. 208-215, September 1976.

16. Fairley, R.E., *Software Engineering Concepts,* McGraw-Hill, pp. 82, 1985.

17. Johnson, S. C., *Yacc: Yet Another Compiler-Compiler,* Bell Laboratories, Murray Hill, New Jersey, July 1978.

18. Lesk, M. E. and Schmidt, E., *Lex - A Lexical Analyzer Generator,* Bell Laboratories, Murray Hill, New Jersey, July 1978.

19. Emdi, Bulent, *The Implementation of a Network CODASYL-DML Interface for the Multi-Lingual Database System,* M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1985.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5100	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	2
4.	Curricular Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943-5100	1
5.	Professor David K. Hsiao, Code 52Hq Computer Science Department Naval Postgraduate School Monterey, California 93943-5100	2
6.	Steven A. Demurjian, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943-5100	2
7.	Helen T. Ritchie 4221 Chaucer Ln. Columbus, Ohio 43220	1
8.	Delbert E. Black 491 North 150 East Kaysville, Utah 84037	3
9.	Jacob A. Anthony, Jr. R D 5 Box 5430 Stroudsburg, Pennsylvania 18360	3

216082
Thesis
A5627 Anthony
c.1 The implementation
of a entity-relation-
ship interface for the
multi-lingual database
system.